



HZ BOOKS

计 算 机 科 学 从 书

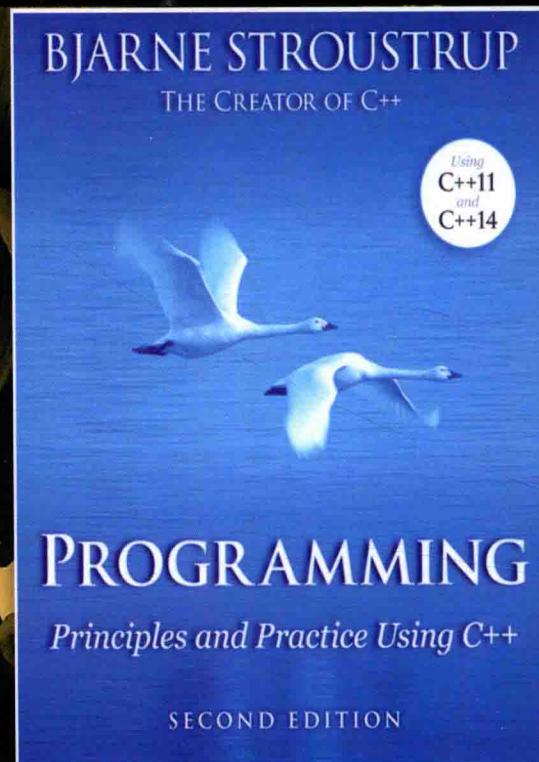
P Pearson

原书第2版

C++程序设计 原理与实践 (基础篇)

[美] 本贾尼·斯特劳斯特鲁普 (Bjarne Stroustrup) 著 任明明 王刚 李忠伟 译

Programming
Principles and Practice Using C++ Second Edition



机械工业出版社
China Machine Press

原书第2版

TP312

P81=2

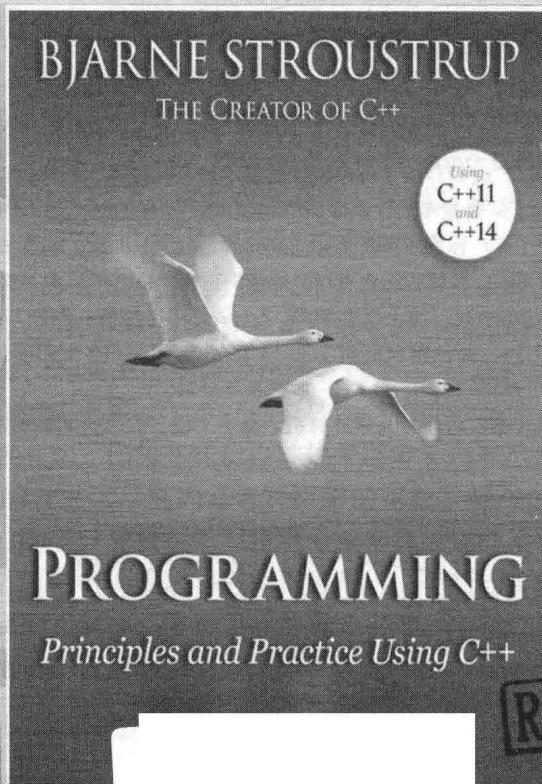
C++设计

原理与实践 (基础篇)

[美] 本贾尼·斯特劳斯特鲁普 (Bjarne Stroustrup) 著 任明明 王刚 李忠伟 译

Programming

Principles and Practice Using C++ Second Edition



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

C++ 程序设计：原理与实践（基础篇）（原书第 2 版）/（美）本贾尼·斯特劳斯特鲁普 (Bjarne Stroustrup) 著；任明明，王刚，李忠伟译。—北京：机械工业出版社，2017.3
(计算机科学丛书)

书名原文：Programming: Principles and Practice Using C++, Second Edition

ISBN 978-7-111-56225-2

I. C… II. ①本… ②任… ③王… ④李… III. C 语言－程序设计－高等学校－教材
IV. TP312.8

中国版本图书馆 CIP 数据核字 (2017) 第 040462 号

本书版权登记号：图字：01-2014-5459

Authorized translation from the English language edition, entitled Programming : Principles and Practice Using C++, Second Edition (978-0-321-99278-9) by Bjarne Stroustrup, published by Pearson Education, Inc., Copyright © 2014.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2017.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内（不包括香港、澳门特别行政区及台湾地区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签，无标签者不得销售。

C++ 之父 Bjarne Stroustrup 的经典著作《C++ 程序设计：原理与实践（原书第 2 版）》基于最新的 C++ 11 和 C++ 14，广泛地介绍了程序设计的基本概念和技术，包括类型系统、算术运算、控制结构、错误处理等；介绍了从键盘和文件获取数值和文本数据的方法以及以图形化方式表示数值数据、文本和几何图形；介绍了 C++ 标准库中的容器（如向量、列表、映射）和算法（如排序、查找和内积）的设计和使用。同时还对 C++ 思想和历史进行了详细的讨论，很好地拓宽了读者的视野。

为方便读者循序渐进地学习，加上篇幅所限，《C++ 程序设计：原理与实践（原书第 2 版）》分为基础篇和进阶篇两册出版，基础篇包括第 1 ~ 11 章、第 17 ~ 19 章和附录 A、C，进阶篇包括第 12 ~ 16 章、第 20 ~ 27 章和附录 B、D、E。本书是基础篇。

本书通俗易懂、实例丰富，可作为大学计算机、电子工程、信息科学等相关专业的教材，也可供相关专业人员参考。

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：和 静

责任校对：李秋荣

印 刷：北京诚信伟业印刷有限公司

版 次：2017 年 4 月第 1 版第 1 次印刷

开 本：185mm×260mm 1/16

印 张：27

书 号：ISBN 978-7-111-56225-2

定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

文艺复兴以来，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的优势，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson, McGraw-Hill, Elsevier, MIT, John Wiley & Sons, Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S.Tanenbaum, Bjarne Stroustrup, Brian W.Kernighan, Dennis Ritchie, Jim Gray, Alfred V.Aho, John E.Hopcroft, Jeffrey D.Ullman, Abraham Silberschatz, William Stallings, Donald E.Knuth, John L.Hennessy, Larry L.Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力相助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专门为本书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们 的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街 1 号

邮政编码：100037



华章教育

华章科技图书出版中心

译者序 |

Programming: Principles and Practice Using C++, Second Edition

程序设计是打开计算机世界大门的金钥匙，它使五花八门的软件对你来说不再是“魔法”。C++语言则是掌握这把金钥匙的有力武器，它优美、高效，从大洋深处到火星表面，从系统核心到高层应用，从掌中的手机到超级计算机，到处都有C++程序的身影。本书的目标不是作为程序设计语言的简单入门教材，而是成为初学者学习基础实用编程技术的绝佳启蒙。如果你愿意努力学习，本书能帮助你理解使用C++语言进行程序设计的基本原理及大量实践技巧，其中大多数可直接用于其他程序设计语言。基于这一目标，注重实践是本书的明显特点。它希望教会你编写真正能被他人所使用的“有用的程序”，而非“玩具程序”。因此，本书不是机械地介绍各种C++特性，而是针对一些具体问题，不断精化其求解方案，在这个过程中自然地引出基本编程技术及相应的C++程序特性。此外，本书还介绍了大量的求解实际问题的程序设计技术，如语法分析器的设计、图形化程序设计、利用正则表达式处理文本、数值计算程序设计以及嵌入式程序设计等。在其他大多数程序设计入门书籍中，是找不到这些内容的。像调试技术、测试技术等其他程序设计书籍着墨不多的话题，本书也有详细的介绍。程序设计远非遵循语法规则和阅读手册那么简单，而在于理解基本思想、原理和技术，并进行大量实践。本书阐述了这一理念，为如何才能达到编写有用的、优美的程序这一最终目标指引了明确的方向。

本书的作者Bjarne Stroustrup是C++语言的设计者和最初的实现者，也是《The C++ Programming Language》(Addison-Wesley出版社)一书的作者。他现在是摩根斯坦利技术部门的总经理和哥伦比亚大学的客座教授，美国国家工程院的院士，ACM会士和IEEE会士。在进入学术界之前，他为AT&T贝尔实验室工作多年。他是ISO标准组织C++委员会的创建者，现在是该委员会语言演化工作组的主席。本书第1版已成为程序设计领域的经典著作，第2版又进行了精心的修订，增加了一些新的内容，包括C++14的一些新特性。

虽然是面向初学者，但本书原版仍是大部头。为方便读者循序渐进地学习，我们重新组织了章节顺序，将原版书组织为基础篇和进阶篇两册。基础篇包括第1~11章、第17~19章和附录A、C，进阶篇包括第12~16章、第20~27章和附录B、D、E。

基础篇	原书	进阶篇	原书
第1章	第1章	第15章	第20章
第2章	第2章	第16章	第21章
第3章	第3章	第17章	第12章
第4章	第4章	第18章	第13章
第5章	第5章	第19章	第14章
第6章	第6章	第20章	第15章
第7章	第7章	第21章	第16章
第8章	第8章	第22章	第22章
第9章	第9章	第23章	第23章
第10章	第10章	第24章	第24章

(续)

基础篇	原书	进阶篇	原书
第 11 章	第 11 章	第 25 章	第 25 章
第 12 章	第 17 章	第 26 章	第 26 章
第 13 章	第 18 章	第 27 章	第 27 章
第 14 章	第 19 章	附录 C	附录 B
附录 A	附录 A	附录 D	附录 D
附录 B	附录 C	附录 E	附录 E

基础篇逻辑上分成四部分：第一部分介绍基本的 C++ 程序设计知识，包括第一个“Hello, World!”程序、对象、类型、值、计算、错误处理、函数、类等内容，以及一个计算器程序实例；第二部分介绍字符方式输入输出，包括输入输出流的基本概念和格式化输出方法；第三部分介绍数据结构的基本知识，重点介绍向量以及自由内存空间、数组、模板和异常；第四部分为附录，介绍了 C++ 语言概要和 Visual Studio 简要入门。通过基础篇的学习，读者可掌握 C++ 最基本的语言特性，以及运用这些特性编写高质量程序的基本技巧。

在此基础上，进阶篇希望帮助读者学习一些更高级的编程技术及相应的 C++ 语言特性，也逻辑上分成四部分：第一部分为数据结构和算法进阶知识，介绍容器和迭代器以及算法和映射；第二部分深入讨论输入输出，介绍图形 /GUI 类和图形化程序设计；第三部分希望拓宽读者的视野，介绍程序设计语言的理念和历史、文本处理技术、数值计算、嵌入式程序设计技术及测试技术，此外还较为详细地介绍了 C 语言与 C++ 的异同；第四部分为附录，包括标准库概要、FLTK 安装以及 GUI 实现等内容。

本书的引言、第 1 章以及第 2 ~ 9 章由任明明翻译，第 10、11 章和第 17 ~ 21 章由李忠伟翻译，第 22 ~ 27 章由刘晓光翻译，第 12 ~ 16 章以及前言、附录等由王刚翻译。翻译大师经典，难度超乎想象。接受任务之初，诚惶诚恐；翻译过程中，如履薄冰；完成后，忐忑不安。虽然竭尽全力，但肯定还有很多错漏之处，敬请读者批评指正。

感谢机械工业出版社华章公司的温莉芳总编辑将此重任交付译者，感谢朱勍等老师为本书所付出的心血，没有她们辛苦的编辑和审校，本书不可能完成。

译者

2016 年 11 月于南开大学

前 言 |

Programming: Principles and Practice Using C++, Second Edition

该死的鱼雷！全速前进。

——Admiral Farragut

程序设计是这样一门艺术，它将问题求解方案描述成计算机可以执行的形式。程序设计中很多工作都花费在寻找求解方案以及对其求精上。通常，只有在真正编写程序求解一个问题的过程中才会对问题本身理解透彻。

本书适合于那些从未有过编程经验但愿意努力学习程序设计技术的初学者，它能帮助读者理解使用 C++ 语言进行程序设计的基本原理并获得实践技巧。本书的目标是使你获得足够多的知识和经验，以便能使用最新、最好的技术进行简单有用的编程工作。达到这一目标需要多长时间呢？作为大学一年级课程的一部分，你可以在一个学期内完成这本书的学习（假定你有另外四门中等难度的课程）。如果你是自学的话，不要期望能花费更少的时间完成学习（一般来说，每周 15 个小时，14 周是合适的学时安排）。

三个月可能看起来是一段很长的时间，但要学习的内容很多。写第一个简单程序之前，就要花费大约一个小时。而且，所有学习过程都是渐进的：每一章都会介绍一些新的有用的概念，并通过真实应用中的例子来阐述这些概念。随着学习进程的推进，你通过程序代码表达思想的能力——让计算机按你的期望工作的能力，会逐渐稳步地提高。我绝不会说：“先学习一个月的理论知识，然后看看你是否能使用这些理论吧。”

为什么要学习程序设计呢？因为我们的文明是建立在软件之上的。如果不理解软件，那么你将退化到只能相信“魔术”的境地，并且将被排除在很多最为有趣、最具经济效益和社会效益的领域之外。当我谈论程序设计时，我所想到的是整个计算机程序家族，从带有 GUI（图形用户界面）的个人计算机程序，到工程计算和嵌入式系统控制程序（如数码相机、汽车和手机中的程序），以及文字处理程序等，在很多日常应用和商业应用中都能看到这些程序。程序设计与数学有些相似，认真去做的话，会是一种非常有用的智力训练，可以提高我们的思考能力。然而，由于计算机能做出反馈，程序设计不像大多数数学形式那么抽象，因而对多数人来说更易接受。可以说，程序设计是一条能够打开你的眼界，将世界变得更美好的途径。最后，程序设计可以是非常有趣的。

为什么学习 C++ 这门程序设计语言呢？学习程序设计是不可能不借助一门程序设计语言的，而 C++ 直接支持现实世界中的软件所使用的那些关键概念和技术。C++ 是使用最为广泛的程序设计语言之一，其应用领域几乎没有局限。从大洋深处到火星表面，到处都能发现 C++ 程序的身影。C++ 是由一个开放的国际标准组织全面考量、精心设计的。在任何一种计算机平台上都能找到高质量的、免费的 C++ 实现。而且，用 C++ 所学到的程序设计思想，大多数可直接用于其他程序设计语言，如 C、C#、Fortran 以及 Java。最后一个原因，我喜欢 C++ 适合编写优美、高效的代码这一特点。

本书不是初学程序设计的最简单入门教材，我写此书的用意也不在此。我为本书设定的目标是——这是一本能让你学到基本的实用编程技术的最简单书籍。这是一个非常雄心勃勃的目标，因为很多现代软件所依赖的技术，不过才出现短短几年时间而已。

我的基本假设是：你希望编写供他人使用的程序，并愿意认真负责地以较高质量完成这个工作，也就是说，假定你希望达到专业水准。因此，我为本书选择的主题覆盖了开始学习实用编程技术所需要的内容，而不只是那些容易讲授和容易学习的内容。如果某种技术是你做好基本编程工作所需要的，那么本书就会介绍它，同时展示用以支持这种技术的编程思想和语言工具，并提供相应的练习，期望你通过做这些练习来熟悉这种技术。但如果你只想了解“玩具程序”，那么你能学到的将远比我所提供的少得多。另一方面，我不会用一些实用性很低的内容来浪费你的时间，本书介绍的内容都是你在实践中几乎肯定用到的。

如果你只是希望直接使用别人编写的程序，而不想了解其内部原理，也不想亲自向代码中加入重要的内容，那么本书不适合你，采用另一本书或另一种程序设计语言会更好些。如果这大概就是你对程序设计的看法，那么请同时考虑一下你从何得来的这种观点，它真的满足你的需求吗？人们常常低估程序设计的复杂程度和它的重要性。我不愿看到，你不喜欢程序设计是因为你的需求与我所描述的软件世界之间不匹配而造成的。信息技术世界中有很多地方是不要求程序设计知识的。本书面向的是那些确实希望编写和理解复杂计算机程序的人。

考虑到本书的结构和注重实践的特点，它也可以作为学习程序设计的第二本书，适合那些已经了解一点 C++ 的人，以及那些会用其他语言编程而现在想学习 C++ 的人。如果你属于其中一类，我不好估计你学习这本书要花费多长时间。但我可以给你的建议是，多做练习。因为你在学习中常见的一个问题是习惯用熟悉的、旧的方式编写程序，而不是在适当的地方采用新技术，多做练习会帮助你克服这个问题。如果你曾经按某种更为传统的方式学习过 C++，那么在进行到第 7 章之前，你会发现一些令你惊奇的、有用的内容。除非你的名字是 Stroustrup，否则你会发现我在本书中所讨论的内容不是“你父辈的 C++”。

学习程序设计要靠编程实践。在这一点上，程序设计与其他需要实践学习的技艺是相似的。你不可能仅仅通过读书就学会游泳、演奏乐器或者开车，必须进行实践。同样，你也不可能不读写大量代码就学会程序设计。本书给出了大量代码实例，都配有说明文字和图表。你需要通过读这些代码来理解程序设计的思想、概念和原理，并掌握用来表达这些思想、概念和原理的程序设计语言的特性。但有一点很重要，仅仅读代码是学不会编程实践技巧的。为此，你必须进行编程练习，通过编程工具熟悉编写、编译和运行程序。你需要亲身体验编程中会出现的错误，学习如何修改它们。总之，在学习程序设计的过程中，编写代码的练习是不可替代的。而且，这也是乐趣所在！

另一方面，程序设计远非遵循一些语法规则和阅读手册那么简单。本书的重点不在于 C++ 的语法，而在于理解基础思想、原理和技术，这是一名好程序员所必备的。只有设计良好的代码才有机会成为一个正确、可靠和易维护的系统的一部分。而且，“基础”意味着延续性：当现在的程序设计语言和工具演变甚至被取代后，这些基础知识仍会保持其重要性。

那么计算机科学、软件工程、信息技术等又如何呢？它们都属于程序设计范畴吗？当然不是！但程序设计是一门基础性的学科，是所有计算机相关领域的基础，在计算机科学领域占有重要的地位。本书对算法、数据结构、用户接口、数据处理和软件工程等领域的重要概念和技术进行了简要介绍，但本书不能替代对这些领域的全面、均衡的学习。

代码可以很有用，同样可以很优美。本书会帮你了解这一点，同时理解优美的代码意味

着什么，并帮你掌握构造优美代码的原理和实践技巧。祝你学习程序设计顺利！

致学生

到目前为止，我在德州农工大学已经用本书教过几千名大一新生，其中 60% 曾经有过编程经历，而剩余 40% 从未见过哪怕一行代码。大多数学生的学习是成功的，所以你也可以成功。

你不一定是在某门课程中学习本书，本书也广泛用于自学。然而，不管你学习本书是作为课程的一部分还是自学，都要尽量与他人协作。程序设计有一个不好的名声——它是一种个人活动，这是不公正的。大多数人在作为一个有共同目标的团体的一份子时，工作效果更好，学习得更快。与朋友一起学习和讨论问题不是“作弊”，而是取得进步最有效同时也是最快乐的途径。如果没有特殊情况的话，与朋友一起工作会促使你表达出自己的思想，这正是测试你对问题理解和确认你的记忆的最有效方法。你没有必要独自解决所有编程语言和编程环境上的难题。但是，请不要自欺欺人——不去完成那些简单练习和大量的习题（即使没有老师督促你，你也不应这样做）。记住：程序设计（尤其）是一种实践技能，需要通过实践来掌握。如果你不编写代码（完成每章的若干习题），那么阅读本书就纯粹是一种无意义的理论学习。

大多数学生，特别是那些爱思考的好学生，有时会对自己努力工作是否值得产生疑问。当你产生这样的疑问时，休息一会儿，重新读一下前言，读一下第 1 章和第 22 章。在那里，我试图阐述我在程序设计中发现了哪些令人兴奋的东西，以及为什么我认为程序设计是能为世界带来积极贡献的重要工具。如果你对我的教学哲学和一般方法有疑问，请阅读引言。

你可能会对本书的厚度感到担心。本书如此之厚的一部分原因是，我宁愿反复重复一些解释说明或增加一些实例，而不是让你自己到处找这些内容，这应该令你安心。另外一个主要原因是，本书的后半部分是一些参考资料和补充资料，供你想要深入了解程序设计的某个特定领域（如嵌入式系统程序设计、文本分析或数值计算）时查阅。

还有，学习中请耐心些。学习任何一种重要的、有价值的新技能都要花费一些时间，而这是值得的。

致教师

本书不是传统的计算机科学导论书籍，而是一本关于如何构造能实际工作的软件的书。因此本书省略了很多计算机科学系学生按惯例要学习的内容（图灵完全、状态机、离散数学、乔姆斯基文法等）。硬件相关的内容也省略了，因为我假定学生从幼儿园开始就已经通过不同途径使用过计算机了。本书也不准备涉及一些计算机科学领域最重要的主题。本书是关于程序设计的（或者更一般地说，是关于如何开发软件的），因此关注的是少量主题的更深入的细节，而不是像传统计算机课程那样讨论很多主题。本书只试图做好一件事，而且计算机科学也不是一门课程可以囊括的。如果本书被计算机科学、计算机工程、电子工程（我们最早的很多学生都是电子工程专业的）、信息科学或者其他相关专业所采用，我希望这门课程能和其他一些课程一起进行，共同形成对计算机科学的完整介绍。

请阅读引言，那里有对我的教学哲学、一般方法等的介绍。请在教学过程中尝试将这些观点传达给你的学生。

ISO 标准 C++

C++ 由一个 ISO 标准定义。第一个 ISO C++ 标准于 1998 年获得批准，所以那个版本的 C++ 被称为 C++98。写本书第 1 版时，我正从事 C++11 的设计工作。最令人沮丧的是，当时我还不能使用一些新语言特性（如统一初始化、范围 for 循环、move 语义、lambda 表达式、concept 等）来简化原理和技术的展示。不过，由于设计该书时考虑到了 C++11，所以很容易在合适的地方添加这些特性。在写作本版时，C++ 标准是 2011 年批准的 C++11，2014 ISO 标准 C++14 中的一些特性正在进入主流的 C++ 实现中。本书中使用的语言标准是 C++11，并涉及少量的 C++14 特性。例如，如果你的编译器不能编译下面的代码：

```
vector<int> v1;
vector<int> v2 {v1}; // C++14 风格的拷贝构造
```

可用如下代码替代：

```
vector<int> v1;
vector<int> v2 = v1; // C++98 风格的拷贝构造
```

若你的编译器不支持 C++11，请换一个新的编译器。好的、现代的 C++ 编译器可从多处下载，见 www.stroustrup.com/compilers.html。使用较早且缺少支持的语言版本会引入不必要的困难。

资源

本书支持网站的网址为 www.stroustrup.com/Programming，其中包含了各种使用本书讲授和学习程序设计所需的辅助资料。这些资料可能会随着时间的推移不断改进，但对于初学者，现在可以找到这些资料：

- 基于本书的讲义幻灯片；
- 一本教师指南；
- 本书中使用的库的头文件和实现；
- 本书中实例的代码；
- 某些习题的解答；
- 可能会有用处的一些链接；
- 勘误表。

欢迎随时提出对这些资料的改进意见。

致谢

我要特别感谢已故的同事和联合导师 Lawrence “Pete” Petersen，很久以前，在我还未感受到教授初学者的惬意时，是他鼓励我承担这项工作，并向我传授了很多能令课程成功的教学经验。没有他，这门课程的首次尝试就会失败。他参与了这门课程最初的建设，本书就是为这门课程所著。他还和我一起反复讲授这门课程，汲取经验，不断改进课程和本书。在本书中我使用的“我们”这个字眼，最初的意思就是指“我和 Pete”。

我要感谢那些直接或间接帮助过我撰写本书的学生、助教和德州农工大学讲授 ENGR 112、113 及 CSCE 121 课程的教师，以及 Walter Daugherity、Hyunyoung Lee、Teresa Leyk、Ronnie Ward、Jennifer Welch，他们也讲授过这门课程。还要感谢 Damian Dechev、Tracy

Hammond、Arne Tolstrup Madsen、Gabriel Dos Reis、Nicholas Stroustrup、J. C. van Winkel、Greg Versoonder、Ronnie Ward 和 Leor Zolman 对本书初稿提出的建设性意见。感谢 Mogens Hansen 为我解释引擎控制软件。感谢 Al Aho、Stephen Edwards、Brian Kernighan 和 Daisy Nguyen 帮助我在夏天躲开那些分心的事，完成本书。

感谢 Art Werschulz，他在纽约福特汉姆大学的课程中使用了本书第 1 版，并据此提出了很多建设性的意见。还要感谢 Nick Maclaren，他在剑桥大学使用了本书的第 1 版，并对本书的习题提出了非常详尽的建议。他的学生在知识背景和专业需求上与德州农工大学大一学生有巨大的差异。

感谢 Addison-Wesley 公司为我安排的审阅者 Richard Enbody、David Gustafson、Ron McCarty 和 K. Narayanaswamy，他们基于自身讲授 C++ 课程或大学计算机科学导论课程的经验，对本书提出了宝贵的意见。还要感谢我的编辑 Peter Gordon 为本书提出的很多有价值的意见以及极大的耐心。我非常感谢 Addison-Wesley 公司的制作团队，他们为本书的高质量出版做出了很多贡献，他们是：Linda Begley（校对），Kim Arney（排版），Rob Mauhar（插图），Julie Nahil（制作编辑），Barbara Wood（文字编辑）。

感谢本书第 1 版的译者，他们发现并帮助澄清了很多问题。特别是，Loïc Joly 和 Michel Michaud 在法语翻译版中做了全面的技术检查，修改了很多问题。

我还要感谢 Brian Kernighan 和 Doug McIlroy 为撰写程序设计类书籍定下了一个非常高的标杆，以及 Dennis Ritchie 和 Kristen Nygaard 为实用编程语言设计提供的非常有价值的经验。

当实际地形与地图不符时，相信实际地形。

——瑞士军队谚语

讲授和学习本书的方法

我们是如何帮助你学习的？又是如何安排学习进程的？我们的做法是，尽力为你提供编写高效的实用程序所需的最基本的概念、技术和工具，包括程序组织、调试和测试、类设计、计算、函数和算法设计、绘图方法（仅介绍二维图形）、图形用户界面（GUI）、文本处理、正则表达式匹配、文件和流输入输出（I/O）、内存管理、科学/数值/工程计算、设计和编程思想、C++ 标准库、软件开发策略、C 语言程序设计技术。认真完成这些内容的学习，我们会学到如下程序设计技术：过程式程序设计（C 语言程序设计风格）、数据抽象、面向对象程序设计和泛型程序设计。本书的主题是程序设计，也就是表达代码意图所需的思想、技术和工具。C++ 语言是我们的主要工具，因此我们比较详细地描述了很多 C++ 语言的特性。但请记住，C++ 只是一种工具，而不是本书的主题。本书是“用 C++ 语言进行程序设计”，而不是“C++ 和一点程序设计理论”。

我们介绍的每个主题都至少出于两个目的：提出一种技术、概念或原理，介绍一个实用的语言特性或库特性。例如，我们用一个二维图形绘制系统的接口展示如何使用类和继承。这使我们节省了篇幅（也节省了你的时间），并且还强调了程序设计不只是简单地将代码拼装起来以尽快地得到一个结果。C++ 标准库是这种“双重作用”例子的主要来源，其中很多主题甚至具有三重作用。例如，我们会介绍标准库中的向量类 `vector`，用它来展示一些广泛使用的设计技术，并展示很多用来实现 `vector` 的程序设计技术。我们的一个目标是向你展示一些主要的标准库功能是如何实现的，以及它们如何与硬件相配合。我们坚持认为一个工匠必须了解他的工具，而不是仅仅把工具当作“有魔力的东西”。

对于一个程序员来说，总是会对某些主题比对其他主题更感兴趣。但是，我们建议你不要预先判断你需要什么（你怎么知道你将来会需要什么呢？），至少每一章都要浏览一下。如果你学习本书是作为一门课程的一部分，你的老师会指导你如何选择学习内容。

我们的教学方法可以描述为“深度优先”，同时也是“具体优先”和“基于概念”。首先， 我们快速地（好吧，是相对快速地，从第 1 章到第 11 章）将一些编写小的实用程序所需的技巧提供给你。在这期间，我们还简明扼要地提出很多工具和技术。我们着重于简单具体的代码实例，因为相对于抽象概念，人们能更快领会具体实例，这就是多数人的学习方法。在最初阶段，你不应期望理解每个小的细节。特别是，你会发现对刚刚还工作得非常好的程序稍加改动，便会呈现出“神秘”的效果。尽管如此，你还是要尝试一下！还有，请完成我们提供的简单练习和习题。请记住，在学习初期你只是没有掌握足够的概念和技巧来准确判断什么是简单的，什么是复杂的。请等待一些惊奇的事情发生，并从中学习吧。

我们会快速通过这样一个初始阶段——我们想尽可能快地带你进入编写有趣程序的阶段。有些人可能会质疑，“我们的进展应该慢些、谨慎些，我们应该先学会走，再学跑！”但

是你见过小孩学习走路吗？实际上小孩在学会平稳地慢慢走路之前就开始尝试跑了。与之相似，你可以先勇猛向前，偶尔摔一跤，从中获得编程的感觉，然后再慢下来，获得必要的精确控制能力和准确的理解。你必须在学会走之前就开始跑！

 你不要投入大量精力试图学习一些语言或技术细节的所有相关内容。例如，你可以熟记所有 C++ 的内置类型及其使用规则。你当然可以这么做，而且这么做会使你觉得自己很博学。但是，这不会使你成为一名程序员。如果你学习中略过一些细节，将来可能偶尔会因为缺少相关知识而被“灼伤”，但这是获取编写好程序所需的完整知识结构的最快途径。注意，我们的这种方法本质上就是小孩学习其母语的方法，也是教授外语的最有效方法。有时你不可避免地被难题困住，我们鼓励你向授课老师、朋友、同事、指导教师等寻求帮助。请放心，在前面这些章节中，所有内容本质上都不困难。但是，很多内容是你所不熟悉的，因此最初可能会感觉有点难。

随后，我们介绍一些入门技巧来拓宽你的知识。我们通过实例和习题来强化你的理解，为你提供一个程序设计的概念基础。

 我们非常强调思想和原理。思想能指导你求解实际问题——可以帮助你知道在什么情况下问题求解方案是好的、合理的。你还应该理解这些思想背后的原理，从而理解为什么要接受这些思想，为什么遵循这些思想会对你和使用你的代码的用户有帮助。没有人会满意“因为事情就是如此”这样的解释。更为重要的是，如果真正理解了思想和原理，你就能将自己已知的知识推广到新的情况；就能用新的方法将思想和工具结合来解决新的问题。知其所以然是学会程序设计技巧所必需的。相反，仅仅不求甚解地记住大量规则和语言特性有很大局限，是错误之源，是在浪费时间。我们认为你的时间很珍贵，尽量不要浪费它。

我们把很多 C++ 语言层面的技术细节放在了附录和手册中，你可以随时按需查找。我们假定你有能力查找到需要的信息，你可以借助目录来查找信息。不要忘了编译器和互联网的在线功能。但要记住，要对所有互联网资源保持足够的怀疑，直至你有足够的理由相信它们。因为很多看起来很权威的网站实际上是由程序设计新手或者想要出售什么东西的人建立的。而另外一些网站，其内容都是过时的。我们在支持网站 www.stroustrup.com/Programming 上列出了一些有用的网站链接和信息。

请不要过于急切地期盼“实际的”例子。我们理想的实例都是能直接说明一种语言特性、一个概念或者一种技术的简短代码。很多现实世界中的实例比我们给出的实例要凌乱很多，而且所能展示的知识也不比我们的实例更多。包含数十万行代码的成功商业程序中所采用的技术，我们用几个 50 行规模的程序就能展示出来。理解现实世界程序的最快途径是好好研究一些基础的小程序。

另一方面，我们不会用“聪明可爱的风格”来阐述我们的观点。我们假定你的目标是编写供他人使用的实用程序，因此书中给出的实例要么是用来说明语言特性，要么是从实际应用中提取出来的。我们的叙述风格都是用专业人员对（将来的）专业人员的那种口气。

一般方法

 本书的内容组织适合从头到尾一章地阅读，当然，你也常常要回过头来对某些内容读上第二遍、第三遍。实际上，这是一种明智的方法，因为当遇到还看不出什么门道的地方时，你通常会快速掠过。对于这种情况，你最终还是会再次回到这个地方。然而，这样做要适度，因为除了交叉引用之外，对本书其他部分，你随便翻开一页，就从那里开始学习，

并希望成功，是不可能的。本书每一节、每一章的内容安排，都假定你已经理解了之前的内容。

本书的每一章都是一个合理的自包含单元，这意味着应一口气读完（当然这只是理论上，实际上由于学生紧密的学习计划，不总是可行）。这是将内容划分为章的主要标准。其他标准包括：从简单练习和习题的角度，每章是一个合适的单元；每一章提出一些特定的概念、思想或技术。这种标准的多样性使得少数章过长，所以不要教条地遵循“一口气读完”的准则。特别是当你已经考虑了思考题，做了简单练习，并做了一些习题时，你通常会发现你需要回过头去重读一些小节和几天前读过的内容。

“它回答了我想到的所有问题”是对一本教材常见的称赞，这对细节技术问题是很理想的，而早期的读者也发现本书有这样的特性。但是，这不是全部的理想，我们希望提出初学者可能想不到的问题。我们的目标是，回答那些你在编写供他人使用的高质量软件时需要考虑的问题。学习回答好的（通常也是困难的）问题是学习如何像一个程序员那样思考所必需的。只回答那些简单的、浅显的问题会使你感觉良好，但无助于你成长为一名程序员。

我们努力尊重你的智力，珍惜你的时间。在本书中，我们以专业性而不是精明伶俐为目标，宁可有节制地表达一个观点而不大肆渲染它。我们尽力不夸大一种程序设计技术或一个语言特性的重要性，但请不要因此低估“这通常是有用的”这种简单陈述的重要程度。如果我们平静地强调某些内容是重要的，意思是如果你不掌握它，或早或晚都会因此而浪费时间。

我们不会伪称本书中的思想和工具是完美的。实际上没有任何一种工具、库、语言或者技术能够解决程序员所面临的所有难题，至多能帮助你开发、表达你解决问题方案而已。我们尽量避免“无害的谎言”，也就是说，我们会尽力避免过于简单的解释，虽然这些解释清晰且易理解，但在实际编程和问题求解时却容易弄错。另一方面，本书不是一本参考手册，如果需要 C++ 详细完整的描述，请参考 Bjarne Stroustrup 的《The C++ Programming Language》第 4 版 (Addison-Wesley 出版社, 2013 年) 和 ISO 的 C++ 标准。

简单练习和习题等

程序设计不仅仅是一种脑力活动，实际动手编写程序是掌握程序设计技巧必不可少的一环。本书提供两个层次的程序设计练习：

- **简单练习：**简单练习是一种非常简单的习题，其目的是帮助学生掌握一些相对死板的实际编程技巧。一个简单练习通常由一系列的单个程序修改练习组成。你应该完成所有简单练习。完成简单练习不需要很强的理解能力、很聪明或者很有创造性。简单练习是本书的基本组成部分，如果你没有完成简单练习，就不能说完成了本书的学习。
- **习题：**有些习题比较简单，有些则很难，但多数习题都是想给学生留下一定的创造和想象空间。如果时间紧张，你可以做少量习题，但题量至少应该能使你弄清楚哪些内容对你来说比较困难，在此基础上应该再多做一些，这是你的成功之道。我们希望本书的习题都是学生能够做出来的，而不是需要超乎常人的智力才能解答的复杂难题。但是，我们还是期望本书习题能给你足够多的挑战，能用光甚至是最好的学生的所有时间。我们不期待你能完成所有习题，但请尽情尝试。

另外，我们建议每个学生都能参与到一个小的项目中去（如果时间允许，能参与更多项

目当然就更好了)。一个项目的目的就是要编写一个完整的有用程序。理想情况下，一个项目由一个多人小组(比如三个人)共同完成。大多数人会发现做项目非常有趣，并在这个过程中学会如何把很多事情组织在一起。

一些人喜欢在读完一章之前就把书扔到一边，开始尝试做一些实例程序；另一些人则喜欢把一章读完后，再开始编码。为了帮助前一种读者，我们用“试一试”板块给出了对于编程实践的一些简单建议。一个“试一试”通常来说就是一个简单练习，而且只着眼于前面刚刚介绍的主题。如果你略过了一个“试一试”而没有去尝试它，那么最好在做这一章的简单练习时做一下这个题目。“试一试”要么是该章简单练习的补充，要么干脆就是其中的一部分。

在每章末尾你都会看到一些思考题，我们设置这些思考题是想为你指出这一章中的重点内容。一种学习思考题的方法是把它们作为习题的补充：习题关注程序设计的实践层面，而思考题则试图帮你强化思想和概念。因此，思考题有点像面试题。

每章最后都有“术语”一节，给出本章中提出的程序设计或C++方面的基本词汇表。如果你希望理解别人关于程序设计的陈述，或者想明确表达出自己的思想，就应该首先弄清术语表中每个术语的含义。

重复是学习的有效手段，我们希望每个重要的知识点都在书中至少出现两次，并通过习题再次强调。

进阶学习

 当你完成本书的学习时，是否能成为一名程序设计和C++方面的专家呢？答案当然是否定的！如果做得好的话，程序设计会是一门建立在多种专业技能上的精妙的、深刻的、需要高度技巧的艺术。你不能期望花四个月时间就成为一名程序设计专家，这与其他学科一样：你不能期望花四个月、半年或一年时间就成为一名生物学专家、一名数学家、一名自然语言(如中文、英文或丹麦文)方面的专家，或是一名小提琴演奏家。但如果你认真地学完了这本书，你可以期待也应该期待的是：你已经在程序设计领域有了一个很好的开始，已经可以写相对简单的、有用的程序，能读更复杂的程序，而且已经为进一步的学习打下了良好的理论和实践基础。

学习完这门入门课程后，进一步学习的最好方法是开发一个真正能被别人使用的程序。在完成这个项目之后或者同时(同时可能更好)学习一本专业水平的教材(如Stroustrup的《The C++ Programming Language》)，学习一本与你做的项目相关的更专门的书(比如，你如果在做GUI相关项目的话，可选择关于Qt的书，如果在做分布式程序的话，可选择关于ACE的书)，或者学习一本专注于C++某个特定方面的书(如Koenig和Moo的《Accelerated C++》、Sutter的《Exceptional C++》或Gamma等人的《Design Patterns》)。完整的参考书目参见本引言或本书最后的参考文献。

 最后，你应该学习另一门程序设计语言。我们认为，如果只懂一门语言，你是不可能成为软件领域的专家的(即使你并不是想做一名程序员)。

本书内容顺序的安排

 讲授程序设计有很多方法。很明显，我们不赞同“我学习程序设计的方法就是最好的学习方法”这种流行的看法。为了方便学习，我们较早地提出一些仅仅几年前还是先进技术的

内容。我们的设想是，本书内容的顺序完全由你学习程序设计过程中遇到的问题来决定，随着你对程序设计的理解和实际动手能力的提高，一个主题一个主题地平滑向前推进。本书的叙述顺序更像一部小说，而不是一部字典或者一种层次化的顺序。

一次性地学习所有程序设计原理、技术和语言功能是不可能的。因此，你需要选择其中一个子集作为起点。更一般地，一本教材或一门课程应该通过一系列的主题子集来引导学生。我们认为，选择适当的主题并给出重点是我们的责任。我们不能简单地罗列出所有内容，必须做出取舍；在每个学习阶段，我们选择省略的内容与选择保留的内容至少同样重要。

作为对照，这里列出我们决定不采用的教学方法（仅仅是一个缩略列表），对你可能有用：

- **C 优先：**用这种方法学习 C++ 完全是浪费学生的时间，学生能用来求解问题的语言功能、技术和库比所需的要少得多，这样的程序设计实践很糟糕。与 C 相比，C++ 能提供更强的类型检查、对新手来说更好的标准库以及用于错误处理的异常机制。
- **自底向上：**学生本该学习好的、有效的程序设计技巧，但这种方法分散了学生的注意力。学生在求解问题过程中所能依靠的编程语言和库方面的支持明显不足，这样的编程实践质量很低、毫无用处。
- **如果你介绍某些内容，就必须介绍它的全部：**这实际上意味着自底向上方法（一头扎进涉及的每个主题，越陷越深）。这种方法硬塞给初学者很多他们并不感兴趣而且可能很长时间内都用不上的技术细节，令他们厌烦。这样做毫无必要，因为一旦学会了编程，你完全可以自己到手册中查找技术细节。这是手册擅长的方面，如果用来学习基本概念就太可怕了。
- **自顶向下：**这种方法对一个主题从基本原理到细节逐步介绍，倾向于把读者的注意力从程序设计的实践层面上转移开，迫使读者一直专注于上层概念，而没有任何机会实际体会这些概念的重要性。这是错误的，例如，如果你没有实际体会到编写程序是那么容易出错，而修正一个错误是那么困难，你就无法体会到正确的软件开发原理。
- **抽象优先：**这种方法专注于一般原理，保护学生不受讨厌的现实问题限制条件的困扰，这会导致学生轻视实际问题、语言、工具和硬件限制。通常，这种方法基于“教学用语言”——一种将来不可能实际应用，有意将学生与实际的硬件和系统问题隔绝开的语言。
- **软件工程理论优先：**这种方法和抽象优先的方法具有与自顶向下方法一样的缺点：没有具体实例和实践体验，你无法体会到抽象理论的价值和正确的软件开发实践技巧。
- **面向对象先行：**面向对象程序设计是一种组织代码和开发工作的很好方法，但并不是唯一有效的方法。特别是，以我们的体会，在类型系统和算法式编程方面打下良好的基础，是学习类和类层次设计的前提条件。本书确实在一开始就使用了用户自定义类型（一些人称之为“对象”），但我们直到第 6 章才展示如何设计一个类，而直到第 17 章才展示了类层次。
- **相信魔法：**这种方法只是向初学者展示强有力的工具和技术，而不介绍其下蕴含的技术和特性。这让学生只能去猜这些工具和技术为什么会有这样的表现，使用它们会付出多大代价，以及它们恰当的应用范围，而通常学生会猜错！这会导致学生过分刻板地遵循相似的工作模式，成为进一步学习的障碍。

自然，我们不会断言这些我们没有采用的方法毫无用处。实际上，在介绍一些特定的内容时，我们使用了其中一些方法，学生能体会到这些方法在这些特殊情况下的优点。但是，当学习程序设计是以实用为目标时，我们不把这些方法作为一般的教学方法，而是采用其他方法：主要是具体优先和深度优先方法，并对重点概念和技术加以强调。

程序设计和程序设计语言

 我们首先介绍程序设计，把程序设计语言放在第二位。我们介绍的程序设计方法适用于任何通用的程序设计语言。我们的首要目的是帮助你学习一般概念、理论和技术，但是这些内容不能孤立地学习。例如，不同程序设计语言在语法细节、编程思想的表达以及工具等方面各不相同。但对于编写无错代码的很多基本技术，如编写逻辑简单的代码（第 5 章和第 6 章），构造不变式（9.4.3 节），以及接口和实现细节分离（9.7 节和 19.1 ~ 19.2 节）等，不同程序设计语言则差别很小。

程序设计技术的学习必须借助于一门程序设计语言，代码设计、组织和调试等技巧是不可能从抽象理论中学到的。你必须用某种程序设计语言编写代码，从中获取实践经验。这意味着你必须学习一门程序设计语言的基本知识。这里说“基本知识”，是因为花几个星期就能掌握一门主流实用编程语言全部内容的日子已经一去不复返了。本书中 C++ 语言相关的内容只是我们选出的它的一个子集，是与编写高质量代码关系最紧密的那部分内容。而且，我们所介绍的 C++ 特性都是你肯定会用到的，因为这些特性要么是出于逻辑完整性的要求，要么是 C++ 社区中最常见的。

可移植性

 编写运行于多种平台的 C++ 程序是很常见的情况。一些重要的 C++ 应用甚至运行于我们闻所未闻的平台！我们认为可移植性和对多种平台架构 / 操作系统的利用是非常重要的特性。本质上，本书的每个例子都不仅是 ISO 标准 C++ 程序，还是可移植的。除非特别指出，本书的代码都能运行于任何一种 C++ 实现，并且确实已经在多种计算机平台和操作系统上测试通过了。

不同系统编译、链接和运行 C++ 程序的细节各不相同，如果每当提及一个实现问题时就介绍所有系统和所有编译器的细节，是非常单调乏味的。我们在附录 B 中给出了 Windows 平台 Visual Studio 和 Microsoft C++ 入门的大部分基本知识。

如果你在使用任何一种流行的但相对复杂的 IDE（集成开发环境，Integrated Development Environment）时遇到了困难，我们建议你尝试命令行工作方式，它极其简单。例如，下面给出的是在 Unix 或 Linux 平台用 GNU C++ 编译器编译、链接和运行一个包含两个源文件 `my_file1.cpp` 和 `my_file2.cpp` 的简单程序所需的全部命令：

```
c++ -o my_program my_file1.cpp my_file2.cpp  
./my_program
```

是的，这真的就是全部。

提示标记

 为了方便读者回顾本书，以及帮读者发现第一次阅读时遗漏的关键内容，我们在页边空白处放置三种“提示标记”：

- ✎: 概念和技术。
- 🖌: 建议。
- ⚠: 警告。

附言

很多章最后都提供了一个简短的“附言”，试图给出本章所介绍内容的全景描述。我们这样做是因为意识到，知识可能是（而且通常就是）令人畏缩的，只有当完成了习题、学习了进一步的章节（应用了本章中提出的思想）并进行了复习之后才能完全理解。不要恐慌，放轻松，这是很自然的，可以预料到的。你不可能一天之内就成为专家，但可以通过学习本书逐步成为一名合格的程序员。学习过程中，你会遇到很多知识、实例和技术，很多程序员已经从中发现了令人激动的和有趣的东西。

程序设计和计算机科学

程序设计就是计算机科学的全部吗？答案当然是否定的！我们提出这一问题的唯一原因就是确实曾有人将其混淆。本书会简单涉及计算机科学的一些主题，如算法和数据结构，但我们的目标还是讲授程序设计：设计和实现程序。这比广泛接受的计算机科学的概念更宽，但也更窄：

- 更宽，因为程序包含很多专业技巧，通常不能归类于任何一种科学。
- 更窄，因为就涉及的计算机科学的内容而言，我们没有系统地给出其基础。

本书的目标是作为一门计算机科学课程的一部分（如果成为一个计算机科学家是你的目标的话），成为软件构造和维护领域第一门课程的基础（如果你希望成为一个程序员或者软件工程师的话），总之是更大的完整系统的一部分。

本书自始至终都依赖计算机科学，我们也强调基本原理，但我们是以理论和经验为基础来讲授程序设计，是把它作为一种实践技能，而不是一门科学。

创造性和问题求解

本书的首要目标是帮助你学会用代码表达自己的思想，而不是教你如何获得这些思想。沿着这样一个思路，我们给出很多实例，展示如何求解问题。每个实例通常先分析问题，随后对求解方案逐步求精。我们认为程序设计本身是问题求解的一种描述形式：只有完全理解了一个问题及其求解方案，你才能用程序来正确表达它；而只有通过构造和测试一个程序，你才能确定你对问题和求解方案的理解是完整、正确的。因此，程序设计本质上是理解问题和求解方案工作的一部分。但是，我们的目标是通过实例而不是通过“布道”或是问题求解详细“处方”的展示来说明这一切。

反馈方法

我们不认为存在完美的教材；个人的需求总是差别很大的。但是，我们愿意尽力使本书和支持材料更接近完美。为此，我们需要大家的反馈，脱离读者是不可能写出好教材的。请大家给我们发送反馈报告，包括内容错误、排版错误、含混的文字、缺失的解释等。我们也感谢有关更好的习题、更好的实例、增加内容、删除内容等的建议。大家提出的建设性意见会帮助将来的读者，我们会将勘误表张贴在支持网站：www.stroustrup.com/Programming。

参考文献

下面列出了前面提及的参考文献，以及可能对你有用的一些文献。

- Becker, Pete, ed. *The C++ Standard*. ISO/IEC 14882:2011.
- Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4, Second Edition*. Prentice Hall, 2008. ISBN 0132354160.
- Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2001. ISBN 0201604647.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2002. ISBN 0201795256.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. “Learning Standard C++ as a New Language.” *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.
- Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 0321958314.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 1999. ISBN 0201615622.

更全面的参考文献列表可以在本书最后找到。

你也许有理由问：“是一些什么人想要教我程序设计？”那么，下面给出作者的一些生平信息。Bjarne Stroustrup 和 Lawrence “Pete” Petersen 合著了本书。Stroustrup 还设计并讲授了面向大学一年级学生的课程，这门课程是与本书同步发展起来的，以本书的初稿作为教材。

Bjarne Stroustrup

我是 C++ 语言的设计者和最初的实现者。在过去大约 40 年间，我使用 C++ 和许多其他程序设计语言进行过各种各样的编程工作。我喜欢那些用在富有挑战性的应用（如机器人控制、绘图、游戏、文本分析以及网络应用）中的优美而又高效的代码。我教过能力和兴趣各异的人设计、编程和 C++ 语言。我是 ISO 标准组织 C++ 委员会的创建者，现在是该委员会语言演化工作组的主席。

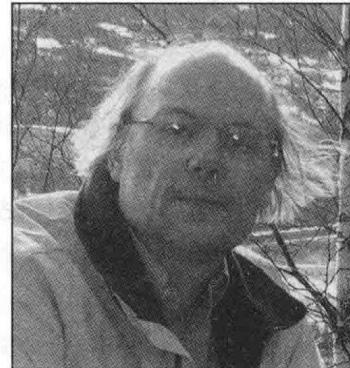
这是我第一本入门性的书。我编著的其他书籍如《The C++ Programming Language》和《The Design and Evolution of C++》都是面向有经验的程序员的。

我生于丹麦奥尔胡斯一个蓝领（工人阶级）家庭，在家乡的大学获得了数学与计算机科学硕士学位。我的计算机科学博士学位是在英国剑桥大学获得的。我为 AT&T 工作了大约 25 年，最初在著名的贝尔实验室的计算机科学研究中心——Unix、C、C++ 及其他很多东西的发明地，后来在 AT&T 实验室研究中心。

我现在是美国国家工程院的院士，ACM 会士（Fellow）和 IEEE 会士。我获得了 2005 年度 Sigma Xi（科学研究协会）的科学成就 William Procter 奖，我是首位获得此奖的计算机科学家。2010 年，我获得了丹麦奥尔胡斯大学最古老也最富声望的奖项 Rigmor og Carl Holst-Knudsens Videnskapspris，该奖项颁发给为科学做出贡献的与该校有关的人士。2013 年，我被位于俄罗斯圣彼得堡的信息技术、力学和光学（ITMO）国立研究大学授予计算机科学荣誉博士学位。

至于工作之外的生活，我已婚，有两个孩子，一个是医学博士，另一个在进行博士后研究。我喜欢阅读（包括历史、科幻、犯罪及时事等各类书籍），还喜欢各种音乐（包括古典音乐、摇滚、蓝调和乡村音乐）。和朋友一起享受美食是我生活中必不可少的一部分，我还喜欢参观世界各地有趣的地方。为了能够享受美食，我还坚持跑步。

关于我的更多信息，请见我的网站 www.stroustrup.com。特别是，你可以在那里找到我名字的正确发音。



Lawrence “Pete” Petersen

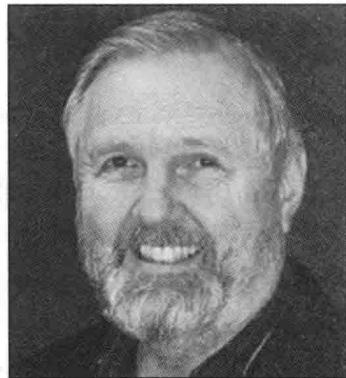
2006 年年末，Pete 如此介绍他自己：“我是一名教师。近 20 年来，我一直在德州农工大学讲授程序设计语言。我已 5 次被学生选为优秀教师，并于 1996 年被工程学院的校友会选为杰出教师。我是 Wakonse 优秀教师计划的委员和教师发展研究院院士。

作为一名陆军军官的儿子，我的童年是在不断迁移中度过的。在华盛顿大学获得哲学学位后，我作为野战炮兵官员和操作测试研究分析员在军队服役了 22 年。1971 年至 1973 年期间，我在俄克拉荷马希尔堡讲授野战炮兵军官的高级课程。1979 年，我帮助创建了测试军官的训练课程，并在 1978 年至 1981 年及 1985 年至 1989 年期间在跨越美国的九个不同地方以首席教官的身份讲授这门课程。

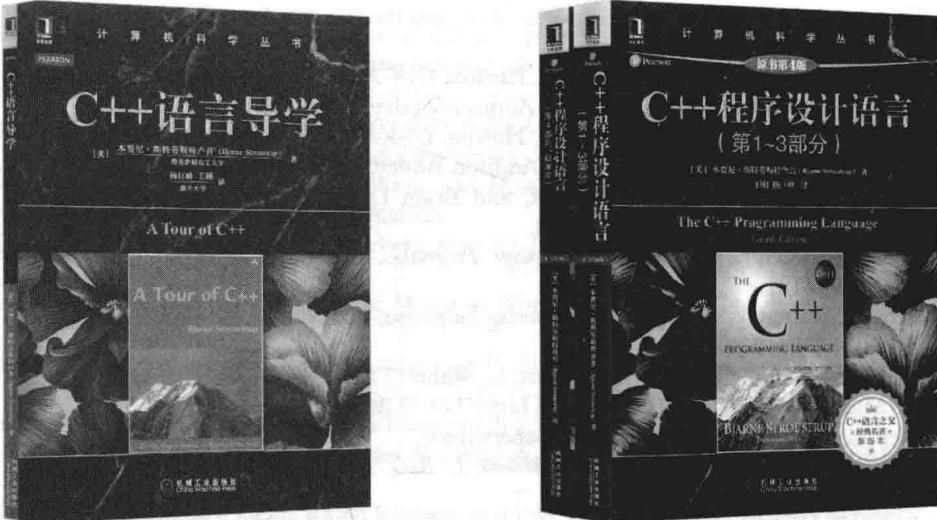
1991 年我组建了一个小型的软件公司，生产供大学院系使用的管理软件，直至 1999 年。我的兴趣在于讲授、设计和实现供人们使用的实用软件。我在乔治亚理工大学获得了工业管理学硕士学位，在德州农工大学获得了教育管理学硕士学位。我还从 NTS 获得了微型计算机硕士学位。我在德州农工大学获得了信息与运营管理学博士学位。

我和我的妻子 Barbara 都生于德州的布莱恩。我们喜欢旅行、园艺和招待朋友；我们花尽可能多的时间陪我们的儿子和他们的家庭，特别是我们的孙子和孙女 Angelina、Carlos、Tess、Avery、Nicholas 和 Jordan。”

令人悲伤的是，Pete 于 2007 年死于肺癌。如果没有他，这门课程绝对不会取得成功。



推荐阅读



C++语言导学

作者: [美] 本贾尼·斯特劳斯卢普 ISBN: 978-7-111-9812-4 定价: 39.00元

本书的目的是让有经验的程序员快速了解C++现代语言。书中几乎涵盖了C++语言的全部核心功能和重要的标准库组件,以简短的篇幅将C++语言的主要特性呈现给读者,并给出一些关键示例,读者可以用很短的时间就能对现代C++的概貌有清晰的了解,尤其是关于面向对象编程和泛型编程的知识。本书没有涉及太多C++语言的细节,非常适合想熟悉C++语言最新特性的C/C++程序设计人员以及精通其他高级语言而想了解C++语言特性的技术人员。

C++程序设计语言 (第1~3部分) (原书第4版)

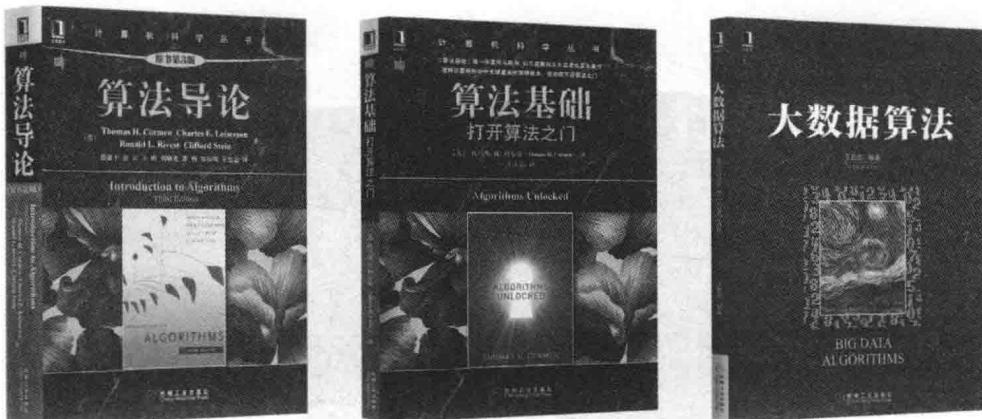
作者: [美] 本贾尼·斯特劳斯卢普 ISBN: 978-7-111-53941-4 定价: 139.00元

C++程序设计语言 (第4部分: 标准库) (原书第4版)

作者: [美] 本贾尼·斯特劳斯卢普 ISBN: 978-7-111-53941-4 定价: 89.00元

本书是在C++语言和程序设计领域具有深远影响、畅销不衰的经典著作,由C++语言的设计者和最初的实现者Bjarne Stroustrup编写,对C++语言进行了最全面、最权威的论述,覆盖标准C++以及由C++所支持的关键编程技术和设计技术。本书英文原版一经面世,即引起业内人士的高度评价和热烈欢迎,先后被翻译成德、希、匈、西、荷、法、日、俄、中、韩等近20种语言,数以百万计的程序员从中受益,是无可取代的C++经典力作。

推荐阅读



算法导论（原书第3版）

作者：Thomas H. Cormen 等 ISBN：978-7-111-40701-0 定价：128.00元

全球超过50万人阅读的算法圣经！算法标准教材，国内外1000余所高校采用

“本书是算法领域的一部经典著作，书中系统、全面地介绍了现代算法：从最快算法和数据结构到用于看似难以解决问题的多项式时间算法；从图论中的经典算法到用于字符匹配、计算集合和数论的特殊算法。本书第3版尤其增加了两章专门讨论van Emde Boas树（最有用的数据结构之一）和多线程算法（日益重要的一个主题）。”

—— Daniel Spielman, 耶鲁大学计算机科学和应用数学Henry Ford II教授

算法基础：打开算法之门

作者：托马斯 H. 科尔曼 ISBN：978-7-111-52076-4 定价：59.00元

《算法导论》第一作者托马斯 H. 科尔曼面向大众读者的算法著作
理解计算机科学中关键算法的简明读本，帮助您开启算法之门

“算法是计算机科学的核心。这是唯一一本力图针对大众读者的算法书籍。它使一个抽象的主题变得简洁易懂，而没有过多拘泥于细节。本书具有深远的影响，还没有人能够比托马斯 H. 科尔曼更能胜任缩小算法专家和公众的差距这一工作。”

—— Frank Dehne, 卡尔顿大学计算机科学系教授

大数据算法

作者：王宏志 ISBN：978-7-111-50849-6 定价：49.00元

本书是国内第一本系统介绍大数据算法设计与分析技术的教材，内容丰富，结构合理，旨在讲述和解决大数据处理和应用中相关算法设计与分析的理论和方法，切实培养读者设计、分析与应用算法解决大数据问题的能力。不仅适合计算机科学、软件工程、大数据、物联网等学科的本科生和研究生使用，而且可供其他相近学科的本科生和研究生使用。同时，该教材还可作为从事大数据相关领域工程技术人员的自学读物。

推荐阅读



深入理解计算机系统（原书第3版）

作者：[美] 兰德尔 E. 布莱恩特 等 译者：龚奕利 等 书号：978-7-111-54493-7 定价：139.00元

理解计算机系统首选书目，10余万程序员的共同选择

卡内基-梅隆大学、北京大学、清华大学、上海交通大学等国内外众多知名高校选用指定教材
从程序员视角全面剖析的实现细节，使读者深刻理解程序的行为，将所有计算机系统的相关知识融会贯通
新版本全面基于X86-64位处理器

基于该教材的北大“计算机系统导论”课程实施已有五年，得到了学生的广泛赞誉，学生们通过这门课程的学习建立了完整的计算机系统的知识体系和整体知识框架，养成了良好的编程习惯并获得了编写高性能、可移植和健壮的程序的能力，奠定了后续学习操作系统、编译、计算机体系结构等专业课程的基础。北大的教学实践表明，这是一本值得推荐采用的好教材。本书第3版采用最新x86-64架构来贯穿各部分知识。我相信，该书的出版将有助于国内计算机系统教学的进一步改进，为培养从事系统级创新的计算机人才奠定很好的基础。

——梅宏 中国科学院院士/发展中国家科学院院士

以低年级开设“深入理解计算机系统”课程为基础，我先后在复旦大学和上海交通大学软件学院主导了激进的教学改革……现在我课题组的青年教师全部是首批经历此教学改革的学生。本科的扎实基础为他们从事系统软件的研究打下了良好的基础……师资力量的补充又为推进更加激进的教学改革创造了条件。

——臧斌宇 上海交通大学软件学院院长

出版者的话	
译者序	
前言	
引言	
作者简介	
第 1 章 计算机、人与程序设计	1
1.1 简介	1
1.2 软件	1
1.3 人	3
1.4 计算机科学	5
1.5 计算机已无处不在	6
1.5.1 有屏幕和无屏幕	6
1.5.2 船舶	6
1.5.3 电信	7
1.5.4 医疗	9
1.5.5 信息领域	10
1.5.6 一种垂直的视角	11
1.5.7 与 C++ 程序设计有何联系	12
1.6 程序员的理想境界	12
思考题	14
术语	15
习题	15
附言	16
第 2 章 Hello, World!	17
2.1 程序	17
2.2 经典的第一个程序	17
2.3 编译	20
2.4 链接	22
2.5 编程环境	22
简单练习	23
思考题	24
术语	25
习题	25
附言	26
第 3 章 对象、类型和值	27
3.1 输入	27
3.2 变量	28
3.3 输入和类型	29
3.4 运算和运算符	31
3.5 赋值和初始化	33
3.5.1 实例：检测重复单词	34
3.6 复合赋值运算符	36
3.6.1 实例：重复单词计数	36
3.7 命名	37
3.8 类型和对象	39
3.9 类型安全	40
3.9.1 安全转换	40
3.9.2 不安全转换	41
简单练习	43
思考题	44
术语	45
习题	45
附言	46
第 4 章 计算	47
4.1 简介	47
4.2 目标和工具	48
4.3 表达式	50
4.3.1 常量表达式	51
4.3.2 运算符	52
4.3.3 类型转换	53
4.4 语句	54
4.4.1 选择语句	55
4.4.2 循环语句	59
4.5 函数	62
4.5.1 为什么使用函数	64
4.5.2 函数声明	65

4.6 vector	65	附言	102
4.6.1 遍历一个 vector	66		
4.6.2 vector 空间增长	67		
4.6.3 一个数值计算实例	67		
4.6.4 一个文本实例	69		
4.7 语言特性	70	第 6 章 编写一个程序	103
简单练习	71	6.1 一个问题	103
思考题	71	6.2 对问题的思考	103
术语	72	6.2.1 程序设计的几个阶段	104
习题	72	6.2.2 策略	104
附言	74	6.3 回到计算器问题	106
第 5 章 错误	75	6.3.1 第一步尝试	106
5.1 简介	75	6.3.2 单词	108
5.2 错误的来源	76	6.3.3 实现单词	109
5.3 编译时错误	77	6.3.4 使用单词	110
5.3.1 语法错误	77	6.3.5 重新开始	111
5.3.2 类型错误	78	6.4 文法	112
5.3.3 警告	78	6.4.1 英文文法	116
5.4 链接时错误	79	6.4.2 设计一个文法	117
5.5 运行时错误	79	6.5 将文法转换为程序	117
5.5.1 调用者处理错误	80	6.5.1 实现文法规则	118
5.5.2 被调用者处理错误	81	6.5.2 表达式	118
5.5.3 报告错误	82	6.5.3 项	121
5.6 异常	83	6.5.4 基本表达式	123
5.6.1 参数错误	84	6.6 试验第一个版本	123
5.6.2 范围错误	85	6.7 试验第二个版本	126
5.6.3 输入错误	86	6.8 单词流	128
5.6.4 窄化错误	88	6.8.1 实现 Token_stream	129
5.7 逻辑错误	89	6.8.2 读单词	130
5.8 估计	91	6.8.3 读数值	131
5.9 调试	92	6.9 程序结构	131
5.9.1 实用调试建议	93	简单练习	133
5.10 前置条件和后置条件	95	思考题	133
5.10.1 后置条件	97	术语	134
5.11 测试	98	习题	134
简单练习	98	附言	135
思考题	99	第 7 章 完成一个程序	136
术语	100	7.1 简介	136
习题	100	7.2 输入和输出	136

7.6 清理代码	143	简单练习	189
7.6.1 符号常量	143	思考题	190
7.6.2 使用函数	145	术语	191
7.6.3 代码布局	145	习题	191
7.6.4 注释	146	附言	192
7.7 错误恢复	148		
7.8 变量	150	第 9 章 类相关的技术细节	193
7.8.1 变量和定义	150	9.1 用户自定义类型	193
7.8.2 引入 name 单词	154	9.2 类和成员	194
7.8.3 预定义名字	156	9.3 接口和实现	194
7.8.4 我们到达目的地了吗	156	9.4 演化一个类	196
简单练习	157	9.4.1 结构和函数	196
思考题	157	9.4.2 成员函数和构造函数	197
术语	158	9.4.3 保持细节私有性	199
习题	158	9.4.4 定义成员函数	200
附言	159	9.4.5 引用当前对象	202
第 8 章 函数相关的技术细节	160	9.4.6 报告错误	202
8.1 技术细节	160	9.5 枚举类型	203
8.2 声明和定义	161	9.5.1 “平坦” 枚举	205
8.2.1 声明的类别	164	9.6 运算符重载	205
8.2.2 变量和常量声明	164	9.7 类接口	206
8.2.3 默认初始化	165	9.7.1 参数类型	207
8.3 头文件	165	9.7.2 拷贝	209
8.4 作用域	167	9.7.3 默认构造函数	209
8.5 函数调用和返回	171	9.7.4 const 成员函数	212
8.5.1 声明参数和返回类型	171	9.7.5 类成员和“辅助函数”	213
8.5.2 返回一个值	172	9.8 Date 类	214
8.5.3 传值	173	简单练习	217
8.5.4 传常量引用	174	思考题	218
8.5.5 传引用	176	术语	218
8.5.6 传值与传引用的对比	178	习题	218
8.5.7 参数检查和转换	179	附言	220
8.5.8 实现函数调用	180		
8.5.9 constexpr 函数	183	第 10 章 输入输出流	221
8.6 计算顺序	184	10.1 输入和输出	221
8.6.1 表达式计算	185	10.2 I/O 流模型	222
8.6.2 全局初始化	186	10.3 文件	223
8.7 名字空间	187	10.4 打开文件	224
8.7.1 using 声明和 using 指令	188	10.5 读写文件	226
		10.6 I/O 错误处理	227

10.7 读取单个值	229	第 12 章 向量和自由空间	267
10.7.1 将程序分解为易管理的子模块	231	12.1 简介	267
10.7.2 将人机对话从函数中分离	233	12.2 <code>vector</code> 的基本知识	268
10.8 用户自定义输出运算符	234	12.3 内存、地址和指针	269
10.9 用户自定义输入运算符	235	12.3.1 <code>sizeof</code> 运算符	271
10.10 一个标准的输入循环	235	12.4 自由空间和指针	272
10.11 读取结构化的文件	236	12.4.1 自由空间分配	273
10.11.1 在内存中的表示	237	12.4.2 通过指针访问数据	274
10.11.2 读取结构化的值	238	12.4.3 指针范围	274
10.11.3 改变表示方法	241	12.4.4 初始化	276
练习	242	12.4.5 空指针	277
思考题	243	12.4.6 自由空间释放	277
术语	243	12.5 析构函数	279
习题	243	12.5.1 生成的析构函数	280
附言	244	12.5.2 析构函数和自由空间	281
第 11 章 定制输入输出	245	12.6 访问元素	282
11.1 有规律的与无规律的输入和输出	245	12.7 指向类对象的指针	283
11.2 格式化输出	245	12.8 类型混用: <code>void*</code> 和类型转换	284
11.2.1 输出整数	246	12.9 指针和引用	285
11.2.2 输入整数	247	12.9.1 指针参数和引用参数	286
11.2.3 输出浮点数	248	12.9.2 指针、引用和继承	287
11.2.4 精度	249	12.9.3 实例: 链表	287
11.2.5 域	250	12.9.4 链表操作	289
11.3 打开和定位文件	250	12.9.5 链表的使用	290
11.3.1 文件打开模式	251	12.10 <code>this</code> 指针	291
11.3.2 二进制文件	252	12.10.1 关于链表使用的更多讨论	293
11.3.3 在文件中定位	254	简单练习	294
11.4 字符串流	254	思考题	294
11.5 面向行的输入	255	术语	295
11.6 字符分类	256	习题	295
11.7 使用非标准分隔符	258	附言	296
11.8 更多未讨论内容	263	第 13 章 向量和数组	297
简单练习	263	13.1 简介	297
思考题	264	13.2 初始化	298
术语	264	13.3 拷贝	299
习题	265	13.3.1 拷贝构造函数	300
附言	266	13.3.2 拷贝赋值	301

13.4 必要的操作	305	14.3.1 类型作为模板参数	333
13.4.1 显式构造函数	307	14.3.2 泛型编程	335
13.4.2 调试构造函数和析构函数	308	14.3.3 概念	336
13.5 访问 <code>vector</code> 元素	309	14.3.4 容器和继承	338
13.5.1 对 <code>const</code> 向量重载运算符	311	14.3.5 整数作为模板参数	338
13.6 数组	311	14.3.6 模板实参推断	340
13.6.1 指向数组元素的指针	312	14.3.7 泛化 <code>vector</code>	340
13.6.2 指针和数组	314	14.4 范围检查和异常	342
13.6.3 数组初始化	316	14.4.1 旁白：设计上的考虑	343
13.6.4 指针问题	316	14.4.2 坦白：使用宏	344
13.7 实例：回文	319	14.5 资源和异常	345
13.7.1 使用 <code>string</code> 实现回文	319	14.5.1 潜在的资源管理问题	346
13.7.2 使用数组实现回文	320	14.5.2 资源获取即初始化	348
13.7.3 使用指针实现回文	321	14.5.3 保证	348
简单练习	321	14.5.4 <code>unique_ptr</code>	349
思考题	322	14.5.5 以移动方式返回结果	350
术语	323	14.5.6 <code>vector</code> 类的 RAII	351
习题	323	简单练习	352
附言	324	思考题	353
第 14 章 向量、模板和异常	325	术语	354
14.1 问题	325	习题	354
14.2 改变大小	327	附言	355
14.2.1 表示方式	327	附录 A C++ 语言概要	356
14.2.2 <code>reserve</code> 和 <code>capacity</code>	328	附录 B Visual Studio 简要入门教程	395
14.2.3 <code>resize</code>	329	术语表	398
14.2.4 <code>push_back</code>	329	参考文献	402
14.2.5 赋值	330		
14.2.6 到目前为止的 <code>vector</code> 类	331		
14.3 模板	332		

计算机、人与程序设计

只有昆虫才专业化。

——R. A. Heinlein

在本章中，我们将介绍一些可以使程序设计变得重要、有意思、富有乐趣的事情。我们还会介绍一些基本的理念与思想。我们希望揭穿几个流行的有关程序设计与程序员的神话。本章是现在可以跳过的内容，当你困扰于一些编程问题并怀疑这一切学习是否值得时，可以返回阅读本章。

1.1 简介

正如大多数的学习一样，学习程序设计就像母鸡和蛋的问题。我们希望开始学习一样东西，但是我们也希望了解为什么学习它。我们想学习一个实用技能，但是也希望确保它不只是暂时的风潮。我们希望自己不是在浪费时间，也不想因夸大的宣传和道德说教而厌烦。现在，你可以只是将本章当作一些有意思的内容来阅读，当你觉得需要更新你大脑中关于“为什么这些技术细节在课堂外很重要”的认知时，再返回来重新阅读本章。

本章陈述了我们的个人见解，阐述了我们认为程序设计中有意思和重要的方面。它解释了激励我们数十年后在这个领域中不断前进的理由。通过阅读本章，你会得到关于“可能的最终目标是什么”以及“程序员可能是哪种人”的一些见解。针对初学者的技术书籍毫无疑问会包含很多基础的内容。在本章中，我们将着眼点从技术细节上移开，考虑一个更大的图景：为什么程序设计是一个有价值的活动？程序设计在人类文明中扮演怎样的角色？程序员在哪些方面所做的贡献值得骄傲？程序设计如何融入软件开发、应用和维护的更大世界中？当人们谈论关于“计算机科学”“软件工程”“信息技术”时，程序设计在其中扮演什么样的角色？程序员是做什么的？一个好的程序员需要具备哪些技能？

对于一个学生来说，理解一个思想、一项技术或一个章节的最紧迫的原因，可能是想以好的成绩通过考试，但是有更多比成绩更重要的东西需要学习！对于那些在软件公司工作的人来说，理解一个思想、一项技术或一个章节的最紧迫的原因，可能是找到一些对目前的项目有帮助的东西，并且不会使控制你的薪水和升职还能解雇你的老板感到恼怒，但同样地，这里有更多值得学习的内容！当我们感到自己的工作会在细微的方面改善人们所生活的世界，我们就会努力将工作做到最好。对于那些需要用几年时间完成的任务（在专业和职业发展中的“事情”），理想和更抽象的思想是决定性的。

我们的文明建立在软件之上。改进软件和发现软件的新用途，是一个人可以改善很多人生活的两种方法。程序设计在这里扮演着一个重要的角色。

1.2 软件

好的软件是看不见的。你不能看到、感觉、称量或敲打它。软件是运行在计算机上的

程序的集合。我们有时候可以看到一台计算机，但我们看到的通常只是包含计算机的一些东西，例如一部电话机、一台照相机、一个面包机、一辆汽车或一台风力涡轮机。我们可以看到软件如何工作。如果软件没有按预想的方式工作，我们会感到困扰或受到伤害。如果软件预想的工作方式不符合我们的需要，我们也会感到困扰或受到伤害。

世界上有多少台计算机？我们不知道，至少有数十亿台。世界上的计算机数量有可能超过人的数量。我们需要统计服务器、桌面计算机、笔记本电脑、平板电脑、智能手机和嵌入式计算机等。

你每天会使用多少台计算机（直接或间接）？在我的汽车中计算机就超过 30 台，移动电话中有 2 台，MP3 播放器中有 1 台，照相机中也有 1 台。我有自己的笔记本电脑（你阅读的这页就是用它写的）与台式计算机。在夏天保持温度与湿度的空调也是 1 台简单的计算机。控制计算机科学系的电梯的也是 1 台计算机。如果你使用的是现代的电视机，其中至少会有 1 台计算机。如果你进行一次网上冲浪，将会通过通信系统接触几十也可能几百台服务器，通信系统中又包含数千台计算机（电话交换机、路由器等）。

我并不是在驾驶一辆后座上带着 30 台笔记本电脑的汽车！重点是这些计算机看起来不像通常的计算机（带有一个屏幕、一个键盘和一个鼠标等），它们作为一个很小的部分嵌入到我们使用的设备中。正因为如此，我的汽车中没有哪个东西看起来像计算机，甚至也没有用于显示地图和行驶方向的屏幕（虽然这在其他车里很常见）。但是，在汽车引擎中会包含很多计算机，用于完成燃油喷射控制与温度监控工作。汽车的助力转向系统包含至少 1 台计算机，广播与安全系统包含多台计算机，我们甚至怀疑车窗的开启 / 关闭都由计算机来控制。新型号的汽车甚至有用于持续检测轮胎气压的计算机。

你日常生活中一天所做的事情需要依赖于多少台计算机？你需要吃饭。如果你生活在一个现代化的城市中，为了将食物提供给你需要巨大的努力，这得益于计划、运输和存储等方面非凡工作。对分布式网络的管理当然是计算机化的，它们之间通过通信系统连接起来。现代化农业也是高度计算机化的，你可以在牛舍附近发现用于监控牛群（年龄、健康、产奶量等）的计算机，农业设备也越来越计算机化，不同政府部门要求的各种表单让老实的农民欲哭无泪。如果出现了事故，你可以在报纸上读到相关报道，当然，报纸上的文章也是通过计算机来书写、进行页面设置以及通过计算机化的设备来印刷的（如果你仍阅读纸质的报纸）——文章通常是以电子形式传输到印刷厂的。书籍的出版采用的是同样的方式。如果你需要上下班，交通流量是通过计算机来监控以避免交通堵塞的（通常是徒劳的）。你喜欢乘坐火车？火车也是计算机化的。有些操作甚至不需要司机来完成，火车的子系统（广播、刹车和票务）包含很多计算机。今天的娱乐业（音乐、电影、电视、舞台表演）也是大量使用计算机的用户。即使非卡通的电影也在大量使用（计算机）动画，音乐和摄影也趋向于数字化存储和传输（使用计算机）。如果你生病，医生为你做检查要使用计算机，病历通常是计算机化的，大多数你遇到的用于治疗的医学仪器也包含计算机。除非你碰巧住在树林里的草屋中，并且不使用任何电气设备（包括电灯），否则你肯定会使用能源。人类发现、提炼、加工和传输石油的过程，从钻头深入地下到本地的汽油（天然气）加油站，整个过程中的每个步骤都要使用计算机。如果你使用信用卡来购买汽油，你也会访问一组计算机。对于煤炭、天然气、太阳能和风力发电，它们都会经过同样的过程。

前面的例子都是“操作上”的，这些计算机设备都直接包含在你所做的事情中。抛开这些不谈，计算机在设计中也起着重要和有趣的作用。你穿的衣服、交谈用的电话和调制自

已喜欢的饮料用的咖啡机，这些都是通过计算机来设计与生产的。优质的现代摄影镜头、精美造型的日常工具和器具，这些几乎都要归功于基于计算机的设计与生产方式。那些设计我们周围环境的工匠、设计师、艺术家和工程师，他们已从很多物理限制中解脱出来，而这些在以前被认为是很本质的局限。如果你生病了，那些用来治愈你的药品也是使用计算机设计的。

最后，科学研究本身严重依赖于计算机。例如用于探秘遥远的恒星的望远镜，我们离开计算机是无法设计、建造和操作它们的，它们产生的大量数据离开计算机也是无法处理的。个别生物学领域的研究人员没有被严重计算机化（不包括照相机、数字录音机、电话的使用），但是回到实验室中，数据要使用计算机模型来存储、分析和检查，并且要和其他科研人员通信。现代化学和生物学（包括医学）大量使用计算机，其程度几年前人们做梦也想不到，并且至今对大多数人仍是难以想象的。人类基因测序是由计算机完成的。让我们描述得更准确一些，人类基因测序是人使用计算机完成的。在所有这些例子中，我们可以看到计算机可以帮助我们完成一些事，而没有计算机很难完成这些事情。

每台计算机都需要运行软件。如果没有软件，计算机就是由硅、金属和塑料组成昂贵的大块头，与门挡、船锚和暖气机没有多大区别。软件中的每行代码都是由人编写的。对于实际执行的每行代码，如果有错的话，就没有什么意义了。但令人惊奇的是，所有代码都正确执行！我们谈论的是用几百种编程语言编写的几十亿行程序代码（程序文本）。让所有这些代码正确运行需要付出惊人的努力和大量技巧。我们希望对所依赖的每种服务和工具进行更多的改进。思考一种你所依赖的服务和工具，你希望看到它们有怎样的改进？至少我们希望服务和工具更小（或更大）、更快速、更可靠、更有特点、更容易使用、更大容量、更好看和更便宜。这些我们想做的改进通常都需要编程。

1.3 人

计算机是人来制造的，也是人来使用的。计算机是一种非常通用的工具，它可以用于很多你无法想象的任务。计算机运行程序，做一些对人有用的事情。换句话说，计算机只是一个硬件，除非某人（程序员）编写代码令它做某些有用的事情。我们常常会忘记软件。更经常忘记程序员。

好莱坞和类似的“流行文化”中的谣言已经给程序员造成很负面的形象。例如，我们总是看到孤独的、肥胖的、丑陋的、不懂社交技巧的讨厌鬼，并且总是痴迷于视频游戏和闯入其他人的计算机。他（几乎总是男人）可能是想毁灭世界，也可能是想拯救世界。很明显，这种漫画式人物的温和版在现实生活中是存在的，但是以我们的经验，在软件开发者中出现这类人的可能性，并不比在律师、警官、汽车销售员、记者、艺术家或政治家中更高。

思考一下你从身边生活中所了解的计算机应用软件。它们是一个孤僻的人在一间黑屋子中独立完成的吗？当然不是，创建一个成功的软件、计算机设备或系统，需要几十、几百乃至几千人扮演一系列令人眼花缭乱的角色，例如程序员、（程序）设计者、测试人员、美工人员、开发小组管理者、实验心理学家、用户界面设计者、分析人员、系统管理员、客户关系人员、音效工程师、项目经理、质量工程师、统计人员、硬件接口工程师、需求分析工程师、安全主管、数学家、销售支持人员、答疑人员、网络设计人员、方法论学家、软件工具管理员、软件库管理员等。这些角色的范围很广，不同组织使用的头衔也不尽相同，这都使人更加迷惑。一个组织中的“工程师”可能是另一个组织中的“程序员”，也可能是另一个

组织中的“开发人员”“技术人员”或“架构师”。甚至有的组织允许其雇员挑选自己的头衔。并不是所有角色都与编程直接相关。但是，对于前面提到的每种角色，我们都曾见到过实际的例子，承担这种角色的人的工作的重要组成部分就是读写代码。另外，一个程序员（扮演这些角色中的一个或多个）在短时期内会和不同应用领域的人打交道，例如生物学家、发动机设计师、律师、汽车销售员、医学研究员、历史学家、地理学家、宇航员、飞机工程师、木材库经理、火箭科学家、保龄球馆建设者、记者和漫画家（这个列表是从个人经历中得到的）。此外，有些人可能在某个阶段是一个程序员，而在职业生涯的其他阶段扮演非程序员的角色。

“程序员是孤立的”的传言完全是杜撰的。那些喜欢独自工作的人通常会选择最可行的工作领域，而且经常痛苦地抱怨“被干扰”或开会的次数。由于现代软件开发是一种团队行为，因此那些喜欢和别人打交道的人会感到更轻松。也就是说，社交和沟通能力是必不可少的，其价值远远高于陈规旧习。在程序员（无论你怎样实际定义一个程序员）必备技能简表中，你会发现沟通能力——与不同背景的人进行良好沟通的能力位列其中，这种沟通包括非正式的、会议形式的、书面形式的和正式报告。我们相信除非你完成过一个或两个团队项目，否则你不会知道什么是编程以及你是否喜欢它。我们喜欢编程的理由是我们会遇到很多友好的、有趣的人，能到各地访问，这是我们职业生涯的一部分。

所有这些的隐含之意是，有各种各样的技能、兴趣和工作习惯的一群人，对开发一个好软件来说是必不可少的。我们的生活质量（有时甚至是生活本身）依赖于这些人。没有人可以扮演我们这里提到的所有角色，明智的人也不会希望扮演每个角色。重点是你的选择范围完全超乎你的想象，你不必局限于特定选择。作为个人，你将“飞”向那些符合你的技能、才智和兴趣的工作领域。

我们谈论的是“程序员”与“编程”，但是编程很明显只是完整画卷的一部分。那些设计船只或移动电话的人不会认为自己是程序员。编程是软件开发中的一个重要部分，但并不是所有工作都是软件开发。类似地，对于大多数产品来说，软件开发是产品开发中的一个重要部分，但并不是所有工作都是产品开发。

⚠ 我们不会假设你（我们的读者）希望成为一个专业程序员，并在剩余的工作生涯中致力于编写代码。即使是那些优秀的程序员——特别是那些最优秀的程序员——也不会将大部分时间用在编写代码上。理解问题需要花费更多的时间，并且通常需要更大的智力投入。当谈及编程的趣味性时，很多程序员都谈到了智力上的挑战这一点。很多优秀的程序员都有（通常意义上）非计算机科学相关专业的学位。例如，如果你进行基因研究方面的软件开发，理解分子生物学将会对你有更大的帮助。如果你进行中世纪文学分析方面的程序设计，阅读一些这类文学著作以及掌握一门或多门相关语言将会对你有更大的帮助。特别是，对于抱有“只关心计算机和编程”态度的人，他们将会难以与那些非程序员的同事交流。这样的人不仅会错过与人交流中最好的那部分（也即生活），而且也会成为不成功的软件开发人员。

那么，我们做何假设呢？编程是一种智力上很具挑战性的技能，是很多重要与有趣的技术方向的一部分。编程也是我们这个世界的重要组成部分，不了解基本的编程知识就像不了解基本的物理、历史、生物或文学知识一样。那些对编程完全无知的人只能退化到相信魔法的境地，对于很多技术角色，这类人是很危险的。如果你看过呆伯特漫画，想象那个尖头发的老板就是你的技术经理，你肯定不愿意遇到他或（更糟糕的是）成为他。另外，编程可以

带来乐趣。

但是，我们如何假设你的编程目的？你也许将编程作为未来学习和工作的重要工具，但不想成为一名专业的程序员。你可能作为一名设计师、作家、经理或科学家，将与其他人进行专业和个人的交流，具有编程方面的基础知识将会使你有一定优势。你也许将专业水平的编程作为你学习和工作的一部分。即使你成为一名专业的程序员，也不意味着你除了编程之外不做任何事。

你可能成为一名计算机工程师或计算机科学家，但即使是这样，你也不会是“时时刻刻编程”。编程是一种用代码表达思想的方式，也是一种帮助问题求解的方式。除非你有值得表达的思想和值得解决的问题，否则编程没有用处（纯粹是浪费时间）。

这是一本关于编程的书，我们承诺帮助你学习如何编程，那么为什么我们会强调非编程的内容和编程的作用的局限性呢？这是因为，一个优秀的程序员会理解代码和编程技术在一个项目中的作用。一个优秀的程序员（在多数情况下）是一个优秀的团队成员，并且会努力理解代码及其产品如何很好地支持整个项目。例如，想象我在为一个新的 MP3 播放器（可能是智能手机或平板电脑的一部分）进行编程，我关心的是代码之美和功能之丰富。我可能一直在大型的、功能强大的计算机上运行自己的代码。我可能会对声音编码理论不屑一顾，因为它们是“与编程无关的”。我将会待在自己的实验室里，而不是走出去与潜在的用户交流，因为“我认为”用户毫无疑问对音乐的品味很差，并且不欣赏图形用户界面（GUI）编程的最新发展。这样做可能给项目带来一场灾难。更大的计算机意味着更昂贵的 MP3 播放器和更短的电池寿命。编码是数字化音乐控制的重要部分，忽视编码技术的发展会导致每首歌曲所需存储空间增加（用不同编码获得相同品质的输出，存储空间的大小差异很大）。无视用户的喜好，不管这喜好在你看来是多么奇怪和过时，通常会导致用户选择其他产品。编写一个好程序的重要环节是理解用户的需求，并且理解这些需求对实现（即代码）的限制。为了完成这幅糟糕程序员的漫画式形象，我还要加上一条——由于对细节的狂热而导致延迟交付和对简单测试的代码正确性的过分自信。我们鼓励你成为一个好程序员，这需要具有广阔的视野，才能创造出好的软件。这些正是对社会的价值的体现和个人满足的关键。

1.4 计算机科学

即使是在最广泛的定义中，也最好将编程看作某些更大事物的一部分。我们可以将编程看作计算机科学、计算机工程、软件工程、信息技术或其他软件相关学科的子学科。我们将编程看作计算机和信息相关科学与工程领域，以及物理学、生物学、医学、历史学、文学和其他学术或研究领域的一种支撑技术。

考虑计算机科学。在 1995 年，美国政府的“蓝皮书”对它的定义如下：“对计算系统和计算的系统研究。这个学科造就的知识体系包含理解计算系统和方法的理论，设计方法、算法和工具，测试概念的方法，分析和验证的方法，以及知识的表示和实现。”正如我们所料，维基百科条目所给出的概念不太正式：“计算机科学或计算科学是对信息和计算的理论基础，以及它们在计算机系统中的实现和应用的研究。计算机科学包含很多子领域，有些强调特定结果的计算（例如计算机图形学），另一些关于计算问题的性质（例如计算复杂度理论）。还有一些集中在实现计算的挑战上。例如，编程语言理论研究描述计算的方法，而计算机编程使用特定的编程语言来解决特定的计算问题。”

编程是一种工具。它是一种表达基础和实践问题求解方案的重要工具，使这些问题可以通过实验来测试、改进，并付诸应用。编程是思想和理论与实际的交汇。这是计算机科学可以成为一种实践训练而不是纯理论，并且影响世界的原因所在。在这方面，和很多其他事情一样，编程必不可少的是实践和理论的良好结合。一定不要退化成单纯的应付了事：只是编写一些代码，满足于用陈旧方式解决当下需求。

1.5 计算机已无处不在

没有人知道关于计算机或软件的所有事。本节内容只是给出一些例子。你也许会看到自己想看的东西。你至少可以理解计算机应用的范围，理解编程远远超出任何个人可以完全掌握的范畴。

大多数人认为计算机只是一个带有显示器和键盘的灰色盒子。这种计算机应该被放置在桌子下面，用于玩游戏、收发消息和邮件、播放音乐。另一些计算机称为笔记本电脑，无聊的商人们在飞机上使用它们查看报表、玩游戏和观看视频。这幅漫画只是冰山的一角。大多数的计算机工作在我们看不到的地方，并且作为维持我们社会运转的系统的一部分。它们中的一些可能充满整个房间，另一些则可能比一枚小的硬币还小。这些有趣的计算机不是通过键盘、鼠标或类似的设备直接与人进行交互的。

1.5.1 有屏幕和无屏幕

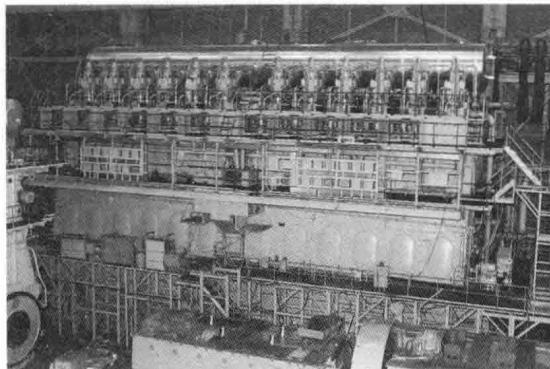
很多人认为计算机是一个相当大的、带有屏幕和键盘的方盒子，并且这种观点很难改变。但是，我们考虑一下这两种计算机：



这两种用于计时的工具本质上都是计算机。实际上，我们猜测它们基本上是带有不同 I/O（输入 / 输出）系统的相同型号的计算机。左边那个驱动一个小的屏幕（与普通计算机的屏幕类似，但是更小），第二个驱动小的电子马达来控制传统的表针和用于表示日期的数字表盘。它们的输入系统都有 4 个按钮（右边那个更容易看清楚）和 1 个无线电接收器，用于与非常精确的“原子”时钟保持同步。这两个计算机的控制程序很多是共享的。

1.5.2 船舶

这两张图片显示的是一台大型的船用柴油机和它可能驱动的巨大的船舶：



我们考虑一下计算机和软件在这里扮演的角色：

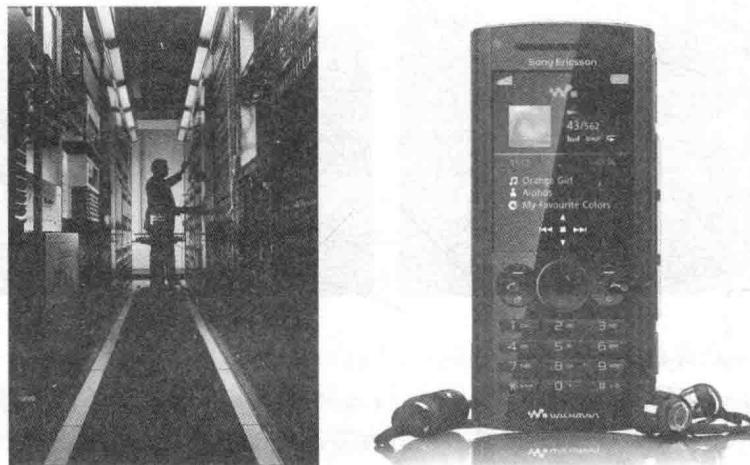
- **设计：**当然，船舶和引擎是使用计算机来设计的。有关用途的列表非常长，主要包括结构和工程制图、一般的计算、空间和零部件的可视化，以及零部件性能的模拟。
- **建造：**现代化的造船厂是高度计算机化的。船舶组装是通过计算机来严格规划的，工作是通过计算机来指导的。焊接是由机器人来完成的。特别是双壳油船，没有小的焊接机器人在壳体之间焊接是无法完成的。那里没有可以容纳人类的空间。为船舶切割钢板是世界上最早的 CAD/CAM（计算机辅助设计和计算机辅助制造）应用之一。
- **引擎：**引擎支持电子燃料喷射，它由数十台计算机控制。对于一台十万马力的引擎（就像照片中的那台），这是一个非凡的任务。例如，引擎管理计算机要持续调节燃料注入，以尽量降低引擎调试不佳导致的污染。很多与引擎相连接的泵（以及船舶的其他部分）本身也是计算机化的。
- **管理：**船舶会航行到某个地方去装卸货物。船队中船只的日程安排是一个持续的过程（当然是计算机化的），这样就可以根据气象、供需情况、港口的空间和吞吐量来调整航线。甚至有网站可以用来查询大型商船在某个时刻的位置。照片中的船舶碰巧是一艘集装箱船（一种世界上最大的船舶，397米长和56米宽），但其他类型的大型现代化船舶也是以相似的方式管理的。
- **监控：**一艘远洋船舶在很大程度上是自治的，它的全体船员可以在到达下一个港口前处理大多数可能产生的紧急事件。但是，它们仍是一个全球网络中的一部分。船员可以访问相当精确的气象信息（通过计算机化的人造卫星）。他们拥有 GPS（全球定位系统）和计算机控制、计算机增强的雷达。如果船员需要休息，大多数系统（包括引擎、雷达等）可以在航线控制室中监控（通过卫星）。如果发现任何不寻常的事或通信连接中断，船员会收到通知。

我们考虑一下在这段简短的介绍中明确提到或暗示的数百台计算机之一出现故障的情况。在第 25 章中，将会对这种情况做出稍微详细的解释。为一艘现代化船舶编写代码是一个很需要技巧的、有趣的行为，也是有用的。海洋运输成本实际上是很低廉的。你在购买那些不在本地生产的东西时会赞赏这点。海洋运输总是比陆地运输更便宜，其主要原因在于计算机和信息的大量使用。

1.5.3 电信

这两张图片显示的是一台电话交换机和一部电话（它碰巧还是一台照相机、一台 MP3

播放器、一台 FM 收音机、一个 Web 浏览器以及更多其他东西):



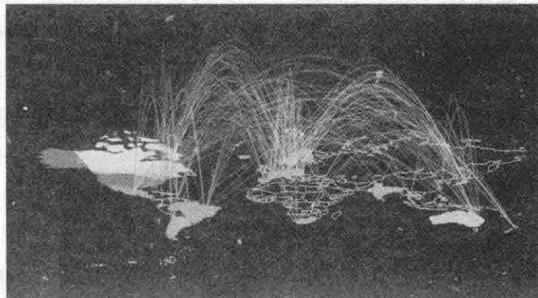
我们考虑一下计算机和软件在这里扮演的角色。你拿起一部电话拨号，你所呼叫的人响应，然后你们可以通话。或者你可能是在发送一条语音留言，可能发送一张由你的电话中的照相机拍摄的照片，或者发送一条文本消息（点击“发送”，由电话完成拨号）。很明显，电话是一台计算机。如果你的电话（像大多数移动电话一样）拥有一个屏幕，并且提供更多传统“老式电话服务”之外的服务，例如 Web 浏览，则它是一台计算机这一点就更加明显。实际上，这类电话通常包含多台计算机：一台用于管理屏幕，一台用于和电话系统通话，可能还会包含更多计算机。

计算机用户最熟悉的可能是电话中管理屏幕、进行 Web 浏览这些部分：它为“所有常见的工作”提供一个图形化用户界面。一部小小的电话在完成其工作时，要与一个庞大系统交互，这是大多数用户不知道、也多半不会怀疑的。我拨打一个德克萨斯州的号码，而这时你正在纽约城度假，但是你的电话铃声在几秒钟内响起，并且我听到你伴着城市交通的嘈杂声说“你好”。很多电话可以在地球上的两个位置之间通话，我们认为这是理所当然的。但我的电话如何找到你的电话？声音如何被传输？声音如何被编码加入数据包？这些问题的答案可以填满比本书更厚的几本书，但是它会涉及分布在相关地理区域中的数百台计算机中的软件和硬件。如果你是不幸的，还会涉及几个通信卫星（它们也是计算机化的）。“不幸”的原因是不能完全补偿进入 2 万英里（1 英里 = 1609.344 米）的太空的代价，光速（决定了你声音传输的速度）是有限的（光纤电缆更好：更短、更快、能传输更多数据）。这些在多数情况下是运转非常好的，骨干通信系统的可靠性可以达到 99.9999%（例如，在 20 年中有 20 分钟断线，断线概率等于 $20 / (20 \times 365 \times 24 \times 60)$ ）。我们遇到的麻烦通常发生在移动电话与最近的主电话交换机之间的通信。

在这里，软件用于在电话之间建立连接，用于将语音编码为数据包通过有线或无线链路传输，用于路由这些消息，用于恢复各种故障，用于持续监控服务的质量和可靠性，当然也用于记账。甚至跟踪系统中所有物理组件，这需要大量智能软件：谁和谁通话？哪部分进入一个新的系统？何时需要进行一些预防性维护？

这个世界的骨干通信系统由很多半独立但互连的系统组成，它可能是最大和最复杂的人工产品。为了更贴近真实情况：记住，这不只是无聊的老式电话带有一些新的铃声或哨音。

在其中已经融入了各种新的基础设施。它们也是 Internet (Web)、金融和贸易系统以及电视台传输电视节目的基础。因此，我们提供另一对图片来说明通信。

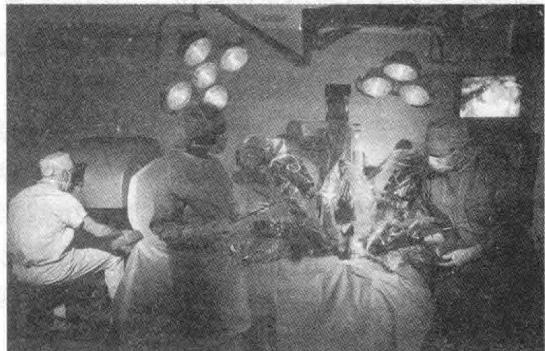


左边的房间是位于纽约华尔街的美国证券交易所的“交易大厅”，右边的图描绘了部分的 Internet 骨干网（一张完整的图将会更加凌乱）。

碰巧，我们也喜欢数字摄影和用计算机绘制特殊地图来可视化知识。

1.5.4 医疗

这两张图片显示的是一台 CAT (计算机轴向断层) 扫描仪和一间计算机辅助手术 (也称为“机器人辅助手术”或“机器人手术”) 的手术室：



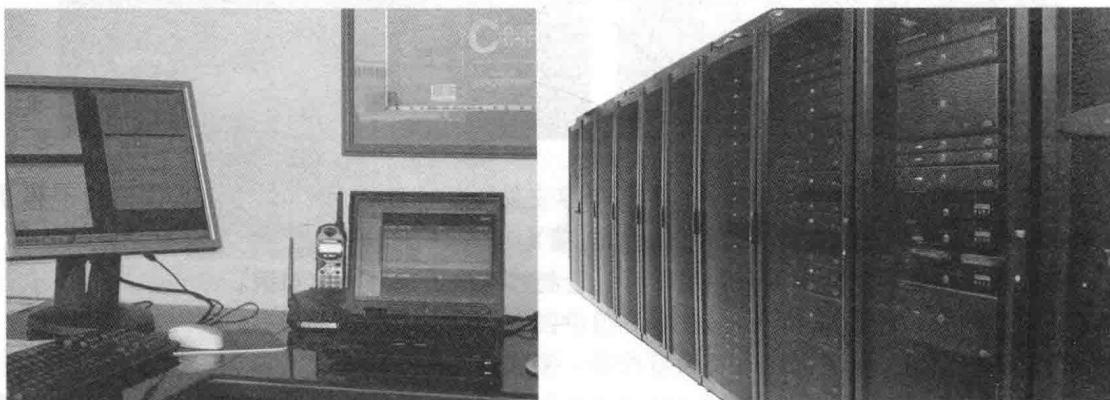
我们考虑一下计算机和软件在这里扮演的角色。扫描仪基本上就是计算机，它发出的脉冲由一台计算机来控制，它读取的内容对我们来说是杂乱无章的，除非将其通过复杂的算法转换成我们可以识别的身体相应部分的（三维）图像。为了进行计算机化的手术，我们还必须进行几个步骤。外科医生借助各种成像技术看清患者的身体内部，从而使手术的部位看起来显著增大并且更明亮。外科医生通过计算机辅助可以使人手不能控制的微小工具，或到达不剖开人体的情况下人手就无法到达的位置。微创手术（腹腔镜手术）是最简单的例子，它减少了数百万人的痛苦和恢复时间。计算机可以帮助稳定外科医生的“手”，以便完成正常情况下不可能的更细致的工作。最后，“机器人”系统可以被远程操作，因此医生有可能远程（通过 Internet）帮助某人。计算机和编程是难以置信的、复杂和有趣的。用户界面、设备控制、成像技术中的每项，都足以使数千名研究人员、工程师和程序员忙碌几十年。

我们听到很多医生关于哪种新的工具对他们的工作最有帮助的讨论：CAT 扫描仪？

MRI 扫描仪？自动血液分析仪？高分辨率超声波仪？PDA？在经过讨论以后，令人惊讶的“胜利者”从这场“竞争”中出现：即时访问病历。了解患者的医疗史（早期疾病、早期用药、过敏、遗传问题、一般健康状况、当前用药等）会简化诊断问题，减少发生错误的机会。

1.5.5 信息领域

这两张图片显示的是一台普通 PC（好吧，是两台）和服务器机群的一部分：



我们曾经将注意力集中在一些“小工具”上，这是出于惯常的原因：你不能看到、感觉到或听到软件。我们不能提供给你一张程序的图片，因此我们展示的是运行软件的“小工具”。但是，很多软件直接处理“信息”。因此，让我们来考虑一下运行“普通软件”的“普通计算机”的“普通用途”。

一个“服务器机群”是提供 Web 服务的多台计算机的集合。运行世界上最先进计算机机群的组织（如 Google、Amazon 和 Microsoft）并没有提及各自服务器的细节，并且服务器机群的规格也在持续变化（所以你在网上找到的信息大多数都过时了）。但是，这些规格是令人惊奇的，它让你确信编程绝不只是在笔记本电脑上简单的计算几个数而已：

- Google 使用了大约 100 万台服务器（每台都比你的笔记本电脑性能强劲），分散在 25 至 50 个“数据中心”里。
- 每个数据中心基本上是一个仓库，大约有 $60m \times 100m$ 或更大。
- 在 2011 年，《纽约时报》报道 Google 的数据中心消耗的电力大约是 2.6 亿瓦（大约相当于拉斯维加斯的能源消耗）。
- 假设一台服务器是 3GHz 的四核处理器，24GB 内存。这意味着 $12 \times 10^{15} Hz$ 的计算能力（大约每秒 12 000 000 000 000 000 次指令）以及 24×10^{15} 字节的内存（大约 $24 000 000 000 000 000$ 字节），每台服务器可能有 4TB 硬盘，总共的硬盘空间就是 4×10^{18} 字节。

我们可能低估了这些值，当读者读到这段时，这几乎是肯定的。特别是在减少能源消耗方面的努力，使得机器的体系结构向每台服务器更多处理器和每个处理器更多核心方向发展。一个 GB 是 1G 字节，大约是 10^9 个字符。一个 TB 是 1T 字节，等于 1000GB，大约是 10^{12} 个字符。一个 PB 是 1P 字节（ 10^{15} 字节），正在变成更常用的计量单位。这是一个相当极端的例子，但是每个大公司都在 Web 上运行程序，并通过它与用户或消费者进行交互。更多的例子包括 Amazon（销售图书和其他商品）、Amadeus（航空票务和汽车租赁）和 eBay（在

线拍卖)。数以百万计的小公司、组织和个人也存在于 Web 上。它们中的大多数并不运行自己的软件，但是也有很多在运行自己的软件，并且其中很多软件都不简单。

其他更传统、更大规模的计算任务主要涉及结算、订单处理、工资处理、记账、账单处理、库存管理、个人记录、学生记录、病历等，基本上每个组织都需要保存记录(商业和非商业、政府和个人)。这些记录是每个组织的支柱。通过计算机处理这些记录看起来很简单：这些信息(记录)中的大多数只需要存储和检索，只有非常少的部分需要处理。这方面的例子包括：

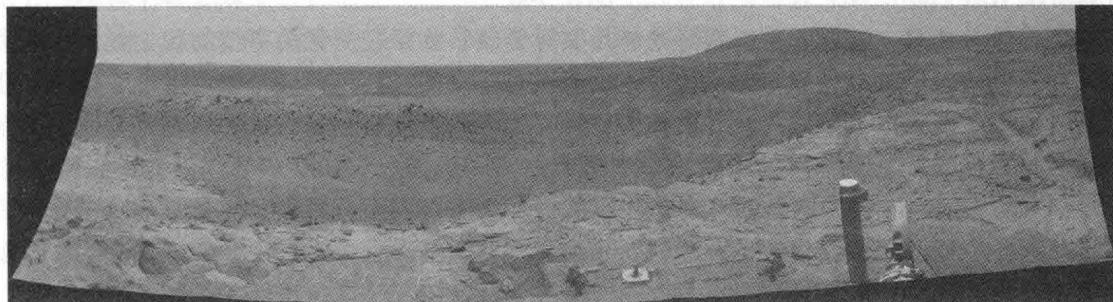
- 12:30 飞往芝加哥的航线是否仍准时？
- Gilbert Sullivan 是否曾经患过麻疹？
- Juan Valdez 订购的咖啡是否已经启运？
- Jack Sprat 在(大约)1996 年购买的是哪种餐椅？
- 2012 年 8 月从 212 区号拨出电话的数量是多少？
- 1 月售出的咖啡壶数量和总价是多少？

规模庞大的数据库使得这些系统非常复杂。针对这种情况，提出了响应更快(对每个查询的响应通常不超过 2 秒钟)和更准确(至少在大多数情况下)的需求。如今，人们谈论 T 字节的数据(一个字节等于用于存储一个普通字符的内存大小)的情形并不少见。这就是传统的“数据处理”，它正在和“Web”相融合，这是由于当前多数的数据库访问都通过 Web 接口。

这种计算机应用通常被称为信息处理。它将重点集中在数据上，特别是大量的数据。这对数据组织和数据传输都提出了挑战，也出现了很多如何以可理解的形式来表示大量数据的有趣工作：“用户接口”是数据处理中的重要方面。例如，考虑分析一部古老的文学作品(比如乔叟的《埃特伯雷故事》或塞万提斯的《堂吉诃德》)，通过比较几十个版本以找出哪个才是作者的实际创作。我们需要根据分析人员提供的多种标准来搜索文本，并且以有助于发现要点的方式来显示结果。提到文本分析，我们就想到了出版：当前，几乎所有的文章、书籍、小册子、报纸等都通过计算机生成。设计出能够很好地支持这一切的软件，对大多数人来说仍是一个缺乏真正好的解决方案的问题。

1.5.6 一种垂直的视角

有人曾经说古生物学家通过研究一块小的骨骼就可以重构一只完整的恐龙，并且描述它的生活方式和自然环境。这有可能是一个夸张的说法，但是这代表了一种思想：通过观察一个简单的物件来思考它暗示了什么。我们考虑一下这张显示火星风景的照片，它由 NASA 的火星漫步者探测器携带的照相机所拍摄：



如果你希望研究“火箭科学”，成为好的程序员是一种方式。各种空间计划需要大量软件设计人员，特别是懂得一些物理、数学、电子工程、机械工程、医疗工程等知识（它们都是载人或非载人空间计划的基础）的人员。两台漫步者火星车成功在火星上运转多年，这是人类文明最伟大的技术胜利之一。其中一台（勇气号）在6年时间里不断发回数据，本书写作时另一台（机遇号）仍在服役，到2014年1月就将在火星上度过第10个年头。而它们的设计寿命只有3个月。

这张照片通过一条通信信道经过每次25分钟的传输延时传输到地球，这里需要很多巧妙的编程和高等数学知识，以便保证以最少的比特数、无差错地传输图片。在地球上，通过某些算法对这张照片进行渲染以恢复颜色和减小失真，这些问题都是由光学传感器和电子传感器引起的。

火星漫步者的控制程序当然也是程序，漫步者每24小时会自动驾驶一次，并执行前一天从地球发送的指令。其中的数据传输是由程序来管理的。

漫步者中的各种计算机使用的操作系统、数据传输和照片重构都是程序，就像用来编写本章的计算机应用程序一样。运行这些程序的计算机是通过CAD/CAM（计算机辅助设计和计算机辅助制造）程序设计和生产的。这些计算机中的芯片是通过计算机化生产线用精密工具组装的，这些工具在它们的设计和制造中也使用计算机（或软件）。对这个很长的组装过程的质量监控涉及很多重要计算。所有这些代码都由程序员用高级编程语言编写，并且通过编译器（本身就是一个程序）转换成机器代码。很多程序使用GUI与用户进行交互，使用输入输出流进行数据交换。

最后，图像处理（包括来自火星漫步者的照片处理）、动画和照片编辑（在互联网上散布着不同版本的标记为“火星人”的漫步者照片）都需要大量编程工作。

1.5.7 与C++程序设计有何联系

这些“多样和复杂的”应用和软件系统与学习编程和使用C++有什么关系？这之间的关系很简单：的确有很多程序员在进行类似的项目。这些事是好的编程可以帮助实现的。本章中用到的每个例子都涉及C++和本书中描述的几种技术。是的，在MP3播放器、船舶、风力发电机组、火星探测和人类基因工程中都会用到C++编程。如果想获得更多的使用C++的例子，你可以查看www.stroustrup.com/applications.html。

1.6 程序员的理想境界

我们希望从自己的程序中获得什么？相对于特定程序的特定功能，一般意义上我们想要什么？我们希望保证正确性，以及可视为正确性的一部分的可靠性。如果程序没有按照设想工作，而且我们还依赖它这种工作方式，那么小则是一个严重干扰，大则是一个危险。我们希望程序设计良好，这样它可以很好地满足实际需要；如果它所做的事情与我们无关，或者以某种我们厌烦的方式完成，则说程序正确是没有意义的。我们同样希望可以负担得起；我可能喜欢用劳斯莱斯汽车或行政专机作为日常交通工具，但是除非我是一名亿万富翁，否则我是负担不起的。

这些是软件（工具、系统）得到非程序员赞赏的外在方面。如果我们希望开发出成功的软件，这些方面必须成为程序员的理想，我们必须将它们永远记在心中，特别是在程序开发的早期阶段。另外，我们还必须关注与代码本身相关的理想：我们的代码必须是可维护的，

那些没有编写它的人可以理解它的结构并进行修改。一个成功的程序可以“生存”很长时
间（经常是几十年），并经过反复修改。例如，它将会移植到新的硬件上，它将会增加新的
功能，它将被修改以使用新的I/O设备（屏幕、视频、音频），它将会用新的自然语言进行交
互等。只有失败的程序才不会被修改。为了保证可维护性，相对于它的需求，程序必须很简
单，而且它的代码必须直接体现要表达的思想。复杂性是简单性和可维护性的敌人，它可以
是问题本身内蕴的（在这种情况下我们不得不处理它），但也可能是由于没有清晰地用代码
表达思想而造成的。我们必须通过良好的编码风格来尽量避免它——编码风格很重要！

这听起来不太难，但是确实很难。为什么？编程从根本上是简单的：就是告诉机器它应
该做什么。但是，为什么编程中要面对很多挑战？计算机根本上也是简单的；它只能做很少
几种操作，例如两个数相加、基于两个数的比较来选择要执行的下一条指令。问题是当我们并不
希望计算机做简单的事情。我们希望“机器”帮助我们做那些难以完成的事，但是计算机是挑
剔的、无情的和不会说话的。更进一步，这个世界要比我们所相信的更复杂，因此我们并未真
正理解自己需求的实际含义。我们只是希望一个程序能够“做这样一些事情”，但是并不希望被技术
细节所困扰。我们通常假设“基本常识”。不幸的是，人们认为很普通的基本常识，在计算机中通常完全不存在（不过在某些特定的、很好理解的情况下，某些精心设计的程序可以模拟它）。

这种思路导致的想法是“编程即理解”：当你能为一个任务编程时，你肯定是理解它了。
反之，当你彻底理解一个任务时，你就可以编写程序去完成它了。换句话说，我们可以将编
程看作努力去彻底理解一个问题的一部分。程序是我们对一个问题的理解的精确表示。

当你在进行编程时，你会花费很多时间尝试理解你试图自动完成的任务。

我们可以将程序开发的过程描述为4个阶段：

- 分析：问题是什么？用户想要什么？用户需要什么？用户可以负担什么？我们需要哪
种可靠性？
- 设计：我们如何解决问题？系统的整体结构将是怎样的？系统包括哪些部分？这些部
分之间如何通信？系统与用户之间如何通信？
- 编程：用代码表达问题的解决方案（设计）。以满足所有约束（时间、空间、金钱、
可靠性等）的方式编写代码。保证这些代码是正确和可维护性的。
- 测试：通过系统化的测试方法确保系统在所要求的所有情况下都正确工作。

编程加上测试通常被称为实现。很明显，将软件开发简单分为四个部分是一种简化。这
四个主题中的每一个都已有很多厚厚的书籍，还有更多的书籍是关于这四个主题之间是如何
关联的。需要注意的是，开发中这几个阶段不是独立的，并且不一定严格按照顺序依次出
现。我们通常从分析开始，但是通过测试的反馈有助于对编程的改进；在努力令程序工作时
遇到的问题可能意味着设计上的问题；在进行设计的过程中可能发现在分析中至今仍被忽
视的某些方面。系统的实际使用通常会暴露分析中的一些弱点。

这里的关键概念是反馈。我们从经验中学习，根据学到的东西改变我们的行为。这是
有效软件开发的根本。对于很多大的项目，我们在开始之前不可能理解有关问题的所有事
情和解决方案。我们可以尝试自己的想法和从编程中得到反馈，但是在开发的早期阶段更容
易（也更快）从写下设计思路、尝试这些设计思路以及朋友的使用中得到反馈。我们知道的
最好的设计工具是黑板（如果你宁愿闻化学气味而不是粉尘，那么你可以使用白板来代替）。
你要尽可能避免独自设计。在已将设计思路解释给其他人之前，不要开始进行编码工作。在

接触键盘之前，与朋友、同事、潜在用户讨论设计和编程技术。令人惊讶的是，仅仅在试图阐明思路的过程中，你就能学到很多东西。最终，程序只不过是对某些思路的表达（用代码）。

同样，当你实现一个程序遇到问题时，将目光从键盘上移开。考虑一下问题本身，而不是你的不完整的方案。与其他交流：解释你希望做什么和为什么它不工作。令人惊讶的是，你向有些人详细解释问题的过程中经常会找到解决方案。除非不得已，不要单独进行调试（找程序错误）。

本书的重点是实现，特别是编程。我们不讲授“如何解决问题”，但提供了足够多的问题实例和它们的解决方案。很多问题的解决是识别出一个已知的问题，并使用其已知的解决技术。只有当大多数子问题以这种方式解决后，你才可能专注于令人兴奋的和有创造性的“跳出固有模式的思维”。因此，我们重点介绍如何用代码清晰表达思想。

用代码直接表达思想是编程的基本理想。这确实是很显然的，但是目前我们还缺乏好的例子来说明这一点。我们将会反复强调这一点。如果我们需要在自己的代码中使用一个整数，我们将它保存在一个 `int` 类型中，它会提供基本的整数操作。如果我们需要使用一个字符串，我们将它保存在一个 `string` 类型中，它会提供基本的文本处理操作。在最基本的层次上，理想情况是每当我们有一个思想、概念、实体、一个我们认为是“事物”的东西、可以写在白板上的东西、在讨论中可以提到的东西、（非计算机科学）教材中讨论的东西时，我们就需要这些东西在程序中作为一个命名实体（类型）存在，并且提供我们认为适合它的操作。如果我们要进行数学计算，我们需要一个 `complex` 类型用于表示复数和一个 `Matrix` 类型用于线性代数计算。如果我们要进行图形处理，我们需要一个 `Shape` 类型、一个 `Circle` 类型、一个 `Color` 类型和一个 `Dialog_box` 类型。当我们处理来自一个温度传感器的数据流时，我们需要一个 `istream` 类型（“i”表示输入）。很明显，每种类型将提供适当的操作，并且只提供适当的操作。这些只是本书中提到的几个例子。除此之外，我们还提供了用于构建用户自定义类型的工具和技术，以便在程序中直接表达你希望体现的概念。

编程是实践和理论相结合的。如果你只重视实践，你将制造出不可扩展、不可维护的程序。如果你只重视理论，你将制造出无法使用的（或无法负担的）玩具程序。

如果你想了解有关编程理想的不同类型的观点，以及一些在编程语言方面对软件做出重要贡献的人，请看第 22 章“理念和历史”。

思考题

思考题的目的是说明本章所解释的关键思想。可将它们看作习题的补充：习题关注的是编程的实践方面，而思考题尝试帮助你阐明思想和概念。在这方面，它们类似于好的面试问题。

1. 什么是软件？
2. 软件为什么重要？
3. 软件哪里重要？
4. 如果有些软件失败，会出现什么问题？列举一些例子。
5. 软件在哪些地方扮演重要角色？列举一些例子。
6. 哪些工作与软件开发相关？列举一些例子。
7. 计算机科学和编程之间的区别是什么？

8. 在船舶的设计、建造和使用中，软件使用在哪些地方？
9. 什么是服务器机群？
10. 你在线提出哪种类型的查询？列举一些例子。
11. 软件在科学方面有哪些应用？列举一些例子。
12. 软件在医疗方面有哪些应用？列举一些例子。
13. 软件在娱乐方面有哪些应用？列举一些例子。
14. 我们期待中的好软件的一般特点有哪些？
15. 一个软件开发者看起来是什么样的？
16. 软件开发的阶段有哪些？
17. 软件开发为什么困难？列举一些原因。
18. 软件的哪些用途为你的生活带来便利？
19. 软件的哪些用途为你的生活带来更多困难？

术语

这些术语是编程和 C++ 方面的基本词汇。如果你希望理解人们谈到的关于编程的主题并阐明自己的想法，你应该知道每个术语的含义。

affordability (负担得起)	customer (客户)	programmer (程序员)
analysis (分析)	design (设计)	programming (编程)
blackboard (黑板)	feedback (反馈)	software (软件)
CAD/CAM (计算机辅助设计 / 制造)	GUI (图形用户界面)	stereotype (陈规旧习)
communication (交流)	ideals (理想境界)	testing (测试)
correctness (正确性)	implementation (实现)	user (用户)

习题

1. 选择一个你最常做的活动（例如上学、吃饭或看电视）。列举计算机直接或间接牵涉其中的方式。
2. 选择一个你最感兴趣或了解的职业。列举这些职业的人涉及计算机的活动。
3. 将你从习题 2 中得到的列表与选择不同职业的朋友交换，并且改进他或她的列表。当你们都完成后，比较你们的结果。记住：这种开放式的练习没有完美的解决方案，永远有可能得到改进。
4. 根据你自己的经验，描述一种离开计算机不可能进行的活动。
5. 列举你已经使用过的程序（软件应用）。列举那些你与程序有明显交互的例子（例如在一台 MP3 播放器中选择一首新歌），而不是那些可能涉及计算机的例子（例如转动你的汽车的方向盘）。
6. 列举十个完全不会涉及（即使是间接的）计算机的人类活动。这可能会比你想得更困难！
7. 列举五个当前没有使用计算机，但是你认为在将来的某个时间会使用的工作。为你选择的每个工作写几句阐述的话。
8. 解释（至少 100 字，但不超过 500 字）为什么你想成为一名计算机程序员。另一方面，如果你相信自己不想成为一名程序员，也请解释。在这两种情况下，请提供深思熟虑、合乎逻辑的论据。

9. 解释（至少 100 字，但不超过 500 字）除了程序员之外，你希望在计算机工业中扮演的角色（不管“程序员”是否是你的首要选择）。
10. 你认为计算机将会发展到有意识、有思想、有能力与人类竞争的程度吗？写一段支持你的观点的话（至少 100 字）。
11. 列举最成功的程序员共有的特点。列举通常认为程序员应该具有的特点。
12. 列举至少五种在本章中提到的计算机程序的应用，并选择一种你最感兴趣和将来某天最想参与的应用。解释为什么选择这种应用（至少 100 字）。
13. 保存本页文字、本章、莎士比亚的所有著作分别需要多大存储空间？假设 1 字节的存储空间可以保存一个字符，试图精确到误差 20% 以内。
14. 你的计算机拥有多大的存储空间？内存呢？磁盘呢？

附言

我们的文明建立在软件之上。软件开发是一种有趣的、对社会有益的并且有利可图的工作，软件领域具有无与伦比的多样性和机遇。当你接触软件，应以有原则和严肃的方式：你要成为解决方案的一部分，而不是增加问题。

我们对遍布在技术文明中的各种软件很敬畏。当然，不是所有的软件应用都是好的，但是这是另外一回事。在这里，我们想强调的是软件是多么普遍，以及我们在日常生活中多么依赖软件。它们都是由像我们这样的人来编写的。所有的科学家、数学家、工程师、程序员等，这些我们简要提到的软件开发者也是像你一样开始的。

现在，让我们回到脚踏实地的学习上，学习那些编程需要的技能。如果你开始怀疑自己努力工作是否值得（大多数有想法的人有时会怀疑它），你可以回过头来重新阅读本章、前言和一点引言的内容。如果你开始怀疑自己是否能掌握这一切，请记住已经有数百万人成为称职的程序员、设计师、软件工程师等。你也可以做到。

Hello, World!

通过写程序来学习编程。

——Brian Kernighan

在本章中，我们给出最简单的 C++ 程序，它实际上可以做任何事。编写此程序的目的如下：

- 让你尝试使用自己的编程环境。
- 给你一个最初的经验——如何让计算机为你做事。

因此，我们提出程序的概念，展示如何使用编译器将程序从人类可读的形式转换到机器指令，并最终在目标平台上执行这些机器指令。

2.1 程序

为了使计算机能够做某件事，你（或其他某人）需要明确告诉它怎么做——细致到那些繁琐的细节。这种对“怎么做”的描述被称为程序，编程是书写和测试程序的行为。

在某种意义上，我们都编过程序。毕竟，我们都曾被告知完成某些任务的步骤，例如“如何开车去最近的电影院”“如何找到楼上的浴室”和“如何用微波炉热饭”。这种描述和程序之间的不同表现在精确度上：人类间的指示常常是不精确的，但我们可以通常委常识加以弥补，但是计算机无法做到。例如，对如何找到楼上的浴室，“沿走廊右转，上楼，它位于你的左边”可能是很好的描述。但是，当你仔细看这些简单的指令，会发现其中语法的草率和指令的不完整。人类很容易做出弥补。例如，假设你坐在桌子旁问浴室的方向。你不需要被告知离开桌子来到走廊、绕过（不是跨过或钻过）桌子、不要踩到猫等。你不需要被告知不要带走刀子和叉子，以及记住打开灯才能看到楼梯。你也不需要被告知进入浴室之前首先要开门。

与此相反，计算机是非常笨拙的。它们做的所有事都要准确、详细地描述。我们考虑“沿走廊右转，上楼，它位于你的左边”。走廊在哪里？什么是走廊？什么是“右转”？什么是楼梯？我如何上楼梯？（每次迈出一步？两步？沿扶手滑上楼梯？）什么在我的左边？它什么时候会在我的左边？为了向计算机精确描述这些“事情”，我们需要一种由特定语法精确定义的语言（对此目标来说英语的结构太过松散了）和针对我们要执行的多种行动精确定义的词汇。这样的语言被称为编程语言，C++ 就是为各种编程任务设计的编程语言。

如果你想知道有关计算机、程序和编程的更多哲学上的细节，请（重新）阅读第 1 章。在本章中，让我们来看一些代码，从一个很简单的程序和运行它的工具和技术开始学习。

2.2 经典的第一个程序

这是经典的第一个程序的一种版本。它在你的屏幕上输出“Hello, World!”：

```
// 程序在屏幕上输出 "Hello, World!"  
  
#include "std_lib_facilities.h"  
  
int main()      // C++ 从 main 函数开始执行  
{  
    cout << "Hello, World!\n";   // 输出 "Hello, World!"  
    return 0;  
}
```

我们可以将这段文字看作交给计算机执行的一组指令，就像我们交给一个厨师的一张菜谱，或我们用于使一个新玩具工作的一组组装指令。我们来讨论这个程序的每行如何工作，从下面这行开始。

```
cout << "Hello, World!\n";      // 输出 "Hello, World!"
```

 这是实际生成输出内容的一行。它打印字符串 `Hello, World!`，并且紧跟着一个换行；也就是说，在打印出 `Hello, World!` 之后，光标将位于下一行的开始位置。光标是一个小的、闪烁的字符或线，它用来显示你可以输入下一个字符的位置。

在 C++ 中，字符串常量是由双引号 ("") 限定的；也就是说，`"Hello, World!\n"` 是一个字符串。`\n` 是一个用于指定换行的“特殊字符”。名称 `cout` 是一个标准的输出流。使用输出操作符 `<<` “输入到 `cout`” 的字符将显示在屏幕上。名称 `cout` 的发音是“see-out”，是“character output stream”的缩写。你在编程时很容易遇到缩写。很自然，在你第一次看到一个缩写还要记住它时会觉得有点儿烦，但是当你开始重复使用一个缩写时，它们将会变得很自然，并且对保持程序文本的简短和可控制是必不可少的。

这行的结尾

```
// 输出 "Hello, World!"
```

是一个注释。在一行中，符号 `//`（符号 / 称为“斜杠”，这里是双斜杠）之后的任何东西都是注释。注释会被编译器忽略，它是写给那些需要阅读代码的人的。在这里，我们使用注释告诉你该行开始部分实际做了什么事情。

注释用于描述程序打算做的事情，它提供的信息通常对人们有用但不能在代码中直接表达。最有可能通过代码中的注释得到帮助的人是你自己——设想你在下个星期或明年回过头来阅读代码，并且忘记了为什么以这种方式书写代码时的情景吧。因此，为你的程序编写好的注释吧。在 7.6.4 中，我们将讨论如何做好注释。

 程序是为两类读者编写的。理所当然，我们编写代码是为了在计算机中执行。但是，程序员也要花费长时间阅读和修改代码。从这个角度，程序员是程序的另一个读者。因此，编写代码也是人与人之间沟通的一种方式。实际上，考虑将人类作为我们的代码的主要读者是有意义的：如果他们（我们）发现代码不容易理解，那么代码就不太可能是正确的。所以，请别忘记代码是用来读的，你应尽最大可能提高其可读性。要注意的是，注释只对人类读者有好处；计算机并不会看注释中的文本。

这个程序中的第一行是一个典型的注释，它简要告诉人类读者这个程序打算做什么：

```
// 程序在屏幕上输出 "Hello, World!"
```

由于代码本身只是说程序在做什么，而不是我们希望它做什么，因此这样的注释是有用的。通常向人类（粗略）解释一个程序将做什么，比我们通过代码向计算机（详细）表达同样的

事情要简洁得多。这种注释通常是我们编写的程序的第一个部分。如无其他情况，它会提醒我们正在尝试做什么。

下一行

```
#include "std_lib_facilities.h"
```

是一个“`#include` 指令”。它指示计算机从名为 `std_lib_facilities.h` 的文件中提供（“包含”）功能。我们编写这个文件的目的是简化使用所有 C++ 实现都具有的功能（“C++ 标准库”）。我们将随着问题的深入解释它的含义。它完全是普通的标准 C++，但是它包含我们不想让你厌烦的细节，这些细节需要另外用许多章才能解释明白。对于这个程序来说，`std_lib_facilities.h` 的重要性表现在我们可以使用标准 C++ 流 I/O 功能。在这里，我们只使用标准输出流 `cout` 和它的输出操作符 `<<`。使用 `#include` 包含的文件通常有后缀 `.h`，它被称为头部或头文件。头文件中包含一些定义，例如在我们的程序中使用的 `cout`。

一台计算机如何知道从哪里开始执行一个程序？它会查找一个称为 `main` 的函数，并且开始执行它在那里找到的指令。这是“Hello, World!”程序的 `main` 函数：

```
int main() // C++ 从 main 函数开始执行
{
    cout << "Hello, World!\n"; // 输出 "Hello, World!"
    return 0;
}
```

每个 C++ 程序必须有一个称为 `main` 的函数，以便告诉计算机从哪里开始执行。一个函数本质上是一个具名的指令序列，计算机会按照指令的编写顺序来执行。一个函数包括 4 个组成部分：

- 一个返回值类型，在这里是 `int`（表示“整数”），它用来指定返回结果的类型。如果有的话，这个函数将会返回该类型的值给调用者。单词 `int` 在 C++ 中是保留词（一个关键字），因此 `int` 不能用作其他东西的名字（见附录 A.3.1）。
- 一个名字，在这里是 `main`。
- 一个参数列表，封闭在一组圆括号中（见 8.2 节和 8.6 节），在这里是 `()`；在这个例子中，参数列表是空的。
- 一个函数体，封闭在一组大括号 `{}` 中，列出了这个函数将要执行的动作（称为语句）。

由此可以看出最简单的 C++ 程序为

```
int main() {}
```

由于这个程序没有做任何事情，因此它并没有多少用处。我们的“Hello, World!”程序的函数 `main()`（“主函数”）中有两个语句：

```
cout << "Hello, World!\n"; // 输出 "Hello, World!"
return 0;
```

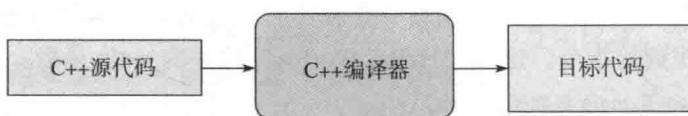
首先，它在屏幕上输出 `Hello, World!`，然后返回一个值 0（零）给它的调用者。由于 `main()` 是由“系统”来调用的，因此我们不会使用返回值。但是，在有些系统（特别是 Unix/Linux）中，返回值可以被用于检查程序是否成功。由 `main()` 返回的零（0）指示该程序成功终止。

在 C++ 程序中，用于描述一个行为的代码，并且它不是一个 `#include` 指令（或其他预

处理器指令，见 4.4 节和附录 A.17)，则被称为一条语句。

2.3 编译

 C++ 是一种编译语言。这意味着要想使一个程序可以运行，首先必须将它从人类可读的格式转换为机器可以“理解”的东西。这个转换过程由一个称为编译器的程序来做。你可以读或写的东西被称为源代码或程序文本，计算机可以执行的东西被称为可执行代码、目标代码或机器代码。典型的 C++ 源代码文件的后缀为 .cpp (例如 `hello_world.cpp`) 或 .h (例如 `std_lib_facilities.h`)，目标代码文件的后缀为 .obj (在 Windows 中) 或 .o (在 Unix 中)。因此，仅用普通单词“代码”是模棱两可的并且会引起混淆；注意只有在可以明确表达含义时再使用它。除非特别说明，我们使用代码来表示“源代码”甚至“不包含注释的源代码”，这是由于注释只是供人类阅读的，在编译器生成目标代码时不会看到它。



编译器会阅读你的源代码，并且尽力理解你所写的内容。编译器会检查你的程序在语法上是否正确，每个单词是否有规定的含义，在程序中是否可以检测到明显的错误，而无须试图实际执行这个程序。你会发现计算机在语法上相当挑剔。忽略我们程序中的有些细节（例如 `#include` 文件、分号或大括号）将会引起错误。与此类似，编译器完全不会容忍拼写错误。我们将通过一系列例子来解释这些，在每个例子中有一个小错误。每个错误是我们经常犯的一种类型错误的例子：

```
// 此处没有 #include 语句
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

我们没有包含任何文件来告诉编译器 `cout` 是什么，因此编译器会抱怨。为了纠正这个错误，让我们增加一个头文件：

```
#include "std_facilities.h"
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

不幸的是，编译器再次抱怨：我们拼写错了 `std_lib_facilities.h`。编译器也不支持以下代码：

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

我们没有用一个 " 来终止字符串。编译器也不支持以下代码：

```
#include "std_lib_facilities.h"
integer main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

在 C++ 中使用缩写 int 而不是单词 integer。编译器也不支持以下代码：

```
#include "std_lib_facilities.h"
int main()
{
    cout < "Hello, World!\n";
    return 0;
}
```

我们使用 < (小于运算符) 而不是 << (输出运算符)。编译器也不支持以下代码：

```
#include "std_lib_facilities.h"
int main()
{
    cout << 'Hello, World!\n';
    return 0;
}
```

我们使用单引号而不是双引号来限制字符串。最后，编译器发现这样的错误：

```
#include "std_lib_facilities.h"
int main()
{
    cout << "Hello, World!\n"
    return 0;
}
```

我们忘记使用分号来终止输出语句。需要注意的是，很多 C++ 语句使用一个分号 (;) 来终止。编译器通过这些分号来识别一个语句在哪里终止，以及另一个语句从哪里开始。关于哪里需要分号，没有简短的、完全正确的、非技术方式的总结。目前来说，请复制我们的应用模式，它可以被归纳为：“在每个没有由右侧大括号 (}) 表示结束的表达式后面放置一个分号。”

为什么我们要花费两页纸的空间和你宝贵的几分钟时间给你看这些带有琐碎错误的小程序的例子呢？像所有的程序员一样，你会花费大量时间在程序源文本中查找错误。在多数时间中，我们看的是包含错误的文本。毕竟，如果确信一些代码是正确的，我们通常会看其他代码或休息一下。令早期的计算机先驱们惊讶的是，他们会造很多代码错误并且不得不花费自己的绝大部分时间来查找它们。现在这也是令大多数的编程新手感到惊讶的地方。

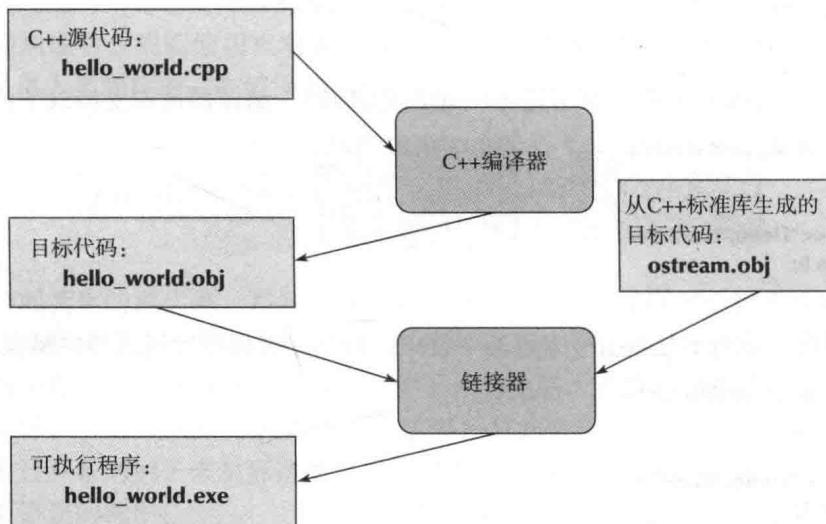
当你在编程时，你有时会对编译器感到相当恼怒。有时候，编译器会抱怨无关紧要的细节（例如缺少一个分号），或者一些你认为“明显正确”的东西。但是，编译器通常是正确的：当它给出一个错误消息并拒绝为你的源代码生成目标代码时，在你的程序中确实有不正确的东西；这意味着你写的程序不符合 C++ 标准的精确定义。

编译器没有任何常识（它不是人类），它对细节是非常挑剔的。由于编译器没有常识，因此你不能让它尝试着去猜测那些“看起来正确”，但是不符合 C++ 定义的东西所表达的意思。假如编译器这样做并且它的猜测与你设想的不同，这时你就需要花费很多时间找出为什么程序没有按你的要求工作。当所有的事都被说清和做到，编译器会将你从大量自己造成的问题中解救出来。

问题中解脱出来。比起由此产生的问题，编译器将我们从更多问题中拯救出来。因此，请记住：编译器是你的朋友，编译器可能是你在编程时最好的朋友。

2.4 链接

程序通常由几个单独的部分组成，它们经常由不同的人来开发。例如，“Hello, World!”程序包含我们编写的部分和 C++ 标准库。这些单独的部分（有时称为编译单元）必须被编译，然后生成的目标代码必须被链接起来以形成一个可执行文件。用于将这些部分链接起来的程序（很自然地）被称为链接器。



请注意目标代码和可执行程序是不能在系统之间移植的。例如，当你为一台 Windows 机器编译时，你得到的支持 Windows 的目标代码无法在 Linux 机器上运行。

一个库是一些代码的集合，它们通常是由其他人编写的，我们通过 `#include` 的文件中的声明来访问库。一个声明是用于指出一段程序如何使用的一条语句，我们将在后面的章节（4.5.2 节）中详细介绍声明。

由编译器发现的错误称为编译时错误，由链接器发现的错误称为链接时错误，直到程序运行时才发现的错误称为运行时错误或逻辑错误。通常来说，编译时错误比链接时错误更容易理解和改正，链接时错误比运行时错误和逻辑错误更容易发现和改正。在第 5 章中，我们将详细讨论这些错误和它们的解决方式。

2.5 编程环境

我们使用编程语言来编写程序。我们使用编译器将自己的源代码转换成目标代码，使用链接器将目标代码链接成一个可执行文件。另外，我们使用一些程序在计算机中输入源代码文本并且编辑它。这些是构成程序员的工具集合或“程序开发环境”中的最基本和最重要的工具。

如果你使用的是命令行窗口，就像很多专业程序员所做的那样，你将不得不亲自发起编译和链接命令。很多专业程序员也使用 IDE（“交互式开发环境”或“集成式开发环境”）。如果你使用的是 IDE，那么简单地点击正确的按钮就可以了。附录 B 介绍了如何在你的

C++ 实现中进行编译和链接。

IDE 通常包含一个编辑器，常具有利用颜色来区分源代码中的注释、关键字和其他部分等有益特性。IDE 还具有其他功能，可帮助来调试代码、编译和运行程序。调试是指在程序中发现错误并排除它们的活动；你在前进的道路上会听到很多有关调试的内容。

学习本书，你可以使用任何一个最新的、符合标准的 C++ 实现。我们所介绍的大多数内容（经过非常小的修改）对所有的 C++ 实现都是正确的，并且代码能运行在任何地方。在我们的工作中，我们使用几种不同的实现。

简单练习

迄今为止，我们讨论了很多有关编程、代码和工具（例如编译器）的内容。现在，你可以运行一个程序。这是本书的一个重点，也是学习编程的一个重点。这是你培养实践技能和好的编程习惯的开始。本章练习的重点在于令你熟悉软件开发环境。当你可以运行“Hello, World!”程序时，你将跨过作为程序员的第一个主要的里程碑。

这个训练的目的是建立或加强你的实际编程技能，以及为你提供编程环境工具方面的经验。典型的训练是对一个独立程序的一系列修改，使之从没有什么价值“发展”成一个实际程序的有用的部分。我们设计了一套传统的练习来测试你的主动性、灵活性或创造性。与此相反，这里的简单练习很少需要你发挥创造力。典型情况下，按顺序完成是非常关键的，每个单独的步骤都很容易（甚至平凡）。请不要自认为聪明，试图跳过某些步骤；这样通常会减慢你的进展或使你感到困惑。

你可能会认为自己理解阅读过的和导师或辅导员告诉你的任何事，但是重复和实践对提高编程能力是很必要的。在这点上，编程和体育、音乐、舞蹈或一些基于技能的行业是相似的。设想一个人在上述任何一个领域竞争，但却不进行常规练习。很容易知道他的表现会怎样。这种持续练习对专业人员意味着终身的长期练习，它是发展和维持高水平实用技能的唯一方式。

因此，不管有多想，也不要跳过这类练习；它们在学习的过程中是非常重要的。现在， 从第一步开始并继续下去，测试每个步骤以确保你做得正确。

如果你不理解所使用的语法的细节也不必担忧，不要害怕向辅导员或朋友寻求帮助。坚持完成所有的练习和部分的习题，所有事情将在适当的时间变得清楚。

因此，这是你的第一个练习：

1. 查看附录 B 并按照这些步骤的要求建立一个工程。建立一个名为 `hello_world` 的空白的 C++ 控制台工程。
2. 输入 `hello_world.cpp`，完全按照下面的内容，将它保存在你的练习目录（文件夹）中，并将它包含在你的 `hello_world` 工程中。

```
#include "std_lib_facilities.h"
int main()    // C++ 从 main 函数开始执行
{
    cout << "Hello, World!\n"; // 输出 "Hello, World!"
    keep_window_open();      // 等待输入一个字符
    return 0;
}
```

在一些 Windows 机器中需要调用 `keep_window_open()`，以防止在有机会阅读输出结果之前窗口被关闭。这是 Windows 的一个特点，而不是 C++ 的。我们在 `std_lib_facilities.`

h 中定义 `keep_window_open()`，以便简化简单文本程序的编写。

你如何找到 `std_lib_facilities.h`？如果你在上课，你可以问辅导员。否则，你可以从我们的支持站点 www.stroustrup.com/Programming 下载它。但是，如果你既没有辅导员又无法访问网站怎么办？（仅）在这种情况下，用下列语句代替 `#include` 语句：

```
#include<iostream>
#include<string>
#include<vector>
#include<algorithm>
#include<cmath>
using namespace std;
inline void keep_window_open() { char ch; cin>>ch; }
```

这里直接使用了标准库，它将会跟随你直到第 5 章，并将在后面章节（8.7 节）中详细解释。

3. 编译和运行“Hello, World!”程序。有些东西很可能不会正常工作。在使用一种新的编程语言或一个新的编程环境进行第一次尝试时很少能正常工作。找到问题并改正它！此处向一个有经验的人寻求帮助是合理的，但是要确定你能够理解别人教给你的，以使自己可独立完成，这样才能更进一步。
4. 现在，你可能遇到一些错误并不得不纠正它。现在是更好地熟悉编译器的错误发现和错误报告功能的时间！尝试来自 2.3 节的 6 个错误，以便查看编程环境对它们的反应。考虑至少 5 个在输入程序时可能发生的其他错误（例如忘记 `keep_window_open()`，在输入单词时按下 Caps Lock 键，或者将分号输入成逗号），并查看在编译和运行每种错误时发生什么。

思考题

这些思考题的基本思路是给你一个机会，以查看你是否注意到并理解了本章的关键点。在回答问题时你可能需要查看正文内容，这是正常和可以预料的。你可能需要重新阅读某一节，这也是正常和可以预料的。但是，如果你需要重新阅读整章或对每个思考题都有问题，你可能需要考虑自己的阅读风格是否有效。你阅读得是否过快？是否应该停下来做某些“试一试”建议？是否应该和朋友一起学习以讨论正文中关于问题的解释？

1. “Hello, World!”程序的目的是什么？
2. 说出一个函数的 4 个部分。
3. 说出必须出现在每个 C++ 程序中的一个函数。
4. 在“Hello, World!”程序中，`return 0;` 这行的目的是什么？
5. 编译器的目的是什么？
6. `#include` 语句的目的是什么？
7. 文件名后缀为 `.h` 在 C++ 中表示什么？
8. 链接器为你的程序做什么？
9. 源文件和目标文件之间的区别是什么？
10. IDE 是什么？它能为你做什么？
11. 即使你理解教材中的所有东西，为什么还是需要实践？

大多数的思考题在它们出现的章节中有明确的回答。但是，我们偶尔会包含某些问题以

提醒你在其他章节中相关的信息，甚至有些内容可能超出本书范围。我们认为这很合理。就编写好的软件和思考这样做的含义而言，那些适合作为独立章节或书籍出现的习题只是很少一部分，前者要复杂得多。

术语

//	executable (可执行的)	main()
<<	function (函数)	object code (目标代码)
C++	header (头文件)	output (输出)
comment (注释)	IDE (集成开发环境)	program (程序)
compiler (编译器)	#include	source code (源代码)
compile-time error (编译时错误)	library (库)	statement (语句)
cout	linker (链接器)	

你或许想以自己的语言逐步发展出一个词汇表。这可以通过对每一章重复下面的习题 5 来完成。

习题

我们将简单练习与习题分别列出；在尝试做习题之前，请先完成该章中的简单练习。这样做将会节约你的时间。

1. 修改程序输出下面两行

Hello, programming!

Here we go!

2. 扩展你曾经学习的知识，编写程序列出令计算机找到楼上浴室的指令（在 2.1 节中讨论）。

你能讨论人类可能假设默认知道而计算机不会的更多的步骤吗？将它们加入你的列表。这是一个“像计算机一样思考”的好的开始。注意，对于大多数人来说，“去浴室”是一个完全充分的指令。对于那些对房子或浴室没有经验的人（想象一个石器时代的人被传送到你的餐厅），这个包含必要指令的列表可能会很长。请不要使用超过一页的指令。为了读者阅读方便，你可以增加一个关于你所想象的房子布局的简短描述。

3. 书写一个有关如何从你的宿舍、公寓、房屋等的前门到你的教室（假设你在参观某个学校；如果你无法想象，选择其他目的地）的门的描述。请一个朋友尝试按照这些指令走，并且让他或她在走的过程中对需要改进之处加以标记。为了保持朋友关系，将这些指令交给一个朋友之前进行“实地测试”是一个好的主意。

4. 找到一本好的菜谱。阅读有关烤制蓝莓松饼的指令（如果你在一个“蓝莓松饼”是一种陌生的异国菜肴的国家，你可以用自己更熟悉的菜肴来代替）。请注意，在得到一点帮助和指令的情况下，这个世界上的大多数人都可以烤出美味的蓝莓松饼。这并不是高级或困难的精细烹饪。但是，对于作者来说，本书中很少有习题像这个一样困难。只是通过一点儿练习，你所能做的事令人吃惊。

- 重新写这些指令使得每个单独的动作在它们自己编号的段落中。认真列出每个步骤使用的所有原料和厨房用具。注意关键的细节，例如理想的烤箱温度、预热烤箱、松饼烤盘的准备、烹调计时的方式，以及将松饼从烤箱中取出时的注意事项。
- 从一个烹调初学者（如果你不是，请你认识的不知道如何烹调的朋友帮助）的角度来考

虑这些指令。填上菜谱作者（几乎可确认是一个有烹调经验的人）省去的很明显的步骤。

- 建立一个烹饪松饼词汇表。（如松饼烤盘是什么？预热做了什么？“烤箱”指的是什么？）
- 现在，烤制一些松饼并享用你的成果。

5. 为“术语”中的每个术语书写一个定义。首先试试看你是否不看本章就可以做到（不太可能），然后浏览本章以找到这些定义。你会发现在初次尝试和本书版本之间有趣的区别。你可以参考一些合适的在线词汇表，例如 www.stroustrup.com/glossary.html。在查找之前先写出自己的定义，你会加强自己从学习中获得的知识。如果你需要重新读某个部分以形成一个定义，这正好可以帮助你来正确理解它。你可以自由使用自己的词汇来进行定义，并且按照你认为合理的细节来完成定义。在通常情况下，主要定义后面跟着一个例子将是有帮助的。你可以将这些定义存储在一个文件中，这样你可以将后面章节的“术语”部分不断增加到这个文件中。

附言

“Hello, World!”程序有多么重要？它的目的是使我们熟悉基本的编程工具。当接触一个新的工具时，我们通常做一个非常简单的例子如“Hello, World!”。在这种方式下，我们将自己所学的东西分为两个部分：首先，我们通过简单程序学习有关工具的基础知识；然后，我们在没有被工具影响的情况下学习更复杂的程序。同时学习工具和语言的难度比先学一种后学另一种要大得多。这种将复杂任务分解成一系列小的（更容易管理的）步骤的学习方式，并不仅仅局限于编程和计算机方面。它在生活中的大多数领域是常见和有用的，特别是在那些需要实用技能的领域。

对象、类型和值

幸运只青睐有准备的头脑。

——Louis Pasteur

本章介绍程序中的数据存储和使用的基础知识。为此，我们首先关注的是从键盘读取数据。在建立了对象、类型、值和变量的基本概念之后，我们介绍几种操作符，并且给出有关 `char`、`int`、`double` 和 `string` 类型的变量使用的例子。

3.1 输入

“Hello, World!” 只是写到屏幕。它产生输出，但不读取任何东西；它也不从用户得到输入。这令人相当厌烦。实际的程序通常基于我们给它的输入产生结果，而不是当我们每次执行它们时只做相同的事。

为了读取某些东西，我们需要从某个地方读入；我们需要在计算机内存中的某个地方放置读取的东西。我们将这样一个“地方”称为一个对象。一个对象是一个具有某种类型的存储区域，类型用来指定可以放置什么样的信息。一个有名字的对象称为一个变量。例如，字符串存放在 `string` 变量中，整数存放在 `int` 变量中。你可以将对象看成一个“盒子”，可以在其中放置该对象类型的数值：

int:	
age:	42

这表示一个名字为 `age` 的 `int` 类型的对象，其中保存的是整型值 42。通过使用一个字符串变量，我们可以从输入中读取一个字符串，然后将它写出来，具体如下：

```
// 读和写一个名字
#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter your first name (followed by 'enter'):\n";
    string first_name;           // first_name 是字符串类型的变量
    cin >> first_name;          // 读取字符串到 first_name
    cout << "Hello, " << first_name << "!\n";
}
```

`#include` 与 `main()` 与第 2 章中的内容相似。由于我们的所有程序（直到第 17 章）都需要 `#include`，我们将不再介绍它以避免分散你的注意力。同样，我们有时展示的只是代码片段，它们只有放入 `main()` 或其他函数中才能工作，比如：

```
cout << "Please enter your first name (followed by 'enter'):\n";
```

我们假设你可以理解如何将这个代码加入一个完整的程序以便测试。

`main()` 中的第一行简单地输出一条信息，提示用户输入一个名字。这个信息通常被称为

一个提示信息，这是由于它提示用户完成某个操作。接下来几行定义了一个名为 `first_name` 的 `string` 变量，读入键盘输入到该变量，并输出一个欢迎词。让我们依次来看这三行：

```
string first_name; // first_name 是字符串类型的变量
```

本行会划分一个可以保存一个字符串的内存区域，并将它命名为 `first_name`：

string:	
first_name:	

在程序中引入一个新的名字并为一个变量分配内存空间的语句被称为定义。

下一行将（键盘）输入的字符串读取到变量：

```
cin >> first_name; // 读取字符串到 first_name
```

名字 `cin` 是由标准库定义的标准输入流（读为“see-in”，“character input”的缩写）。操作符 `>>`（“get from”）的第二个操作指定输入到哪里。因此，如果我们输入某个名字，如 `Nicholas`，紧跟一个新行，那么字符串 `"Nicholas"` 将会变成 `first_name` 的值：

string:	
first_name:	Nicholas

这里换行是必要的，它用来引起计算机的注意。直到输入一个换行（按回车键），计算机才收集这些字符。这段“时间”给你改变主意的机会，在按回车键之前删除或修改某些字符。这个换行不会保存在内存中成为字符串的一部分。

在将输入的字符串放入 `first_name` 之后，我们可以使用它：

```
cout << "Hello, " << first_name << "!\n";
```

本行会在屏幕上打印 `Hello,`，接着是 `Nicholas` (`first_name` 的值)，接着是 `!` 和一个新行 (`\n`)：

Hello, Nicholas!

如果我们喜欢重复和额外的输入，我们可以用三个单独的输出语句来代替：

```
cout << "Hello, ";
cout << first_name;
cout << "!\n";
```

但是，我们不是专业打字员，更重要的是我们非常不喜欢不必要的重复（因为重复更容易出错），因此将三个输出语句合并为一个语句。

注意，我们在 `"Hello,"` 处使用的是引号，而在 `first_name` 处没有这样做。我们使用引号来表示字符串字面值。当不使用引号时，我们输出的是名字对应的值。考虑如下：

```
cout << "first_name" << " is " << first_name;
```

在这里，“`first_name`”输出的是十个字符 `first_name`，而 `first_name` 输出的是变量 `first_name` 对应的值，在这种情况下是 `Nicholas`。因此，我们得到：

first_name is Nicholas

3.2 变量

如果离开存储在内存中的数据，我们基本上不能使用计算机做任何有意义的事，正如上面的例子中所说的字符串输入。我们用来存储数据的“位置”被称为对象。我们需要使用一

个名字来访问一个对象。一个命名后的对象被称为一个变量，它有特定的类型（例如 `int` 或 `string`），类型决定我们将什么赋给对象（例如，123 可以赋给 `int` 型，“Hello, World!\n”可以赋给 `string` 型），以及可以使用的操作（例如，我们可以对 `int` 型使用 * 运算符进行相乘，对 `string` 型使用 <= 操作符进行比较）。我们赋给变量的数据项被称为值。一个用来定义变量的语句（通常）被称为定义，一个定义可以（也通常应该）提供一个初始值。如下：

```
string name = "Annemarie";
int number_of_steps = 39;
```

我们可以像这样来可视化这些变量：

int:	39	string:	Annemarie
number_of_steps:		name:	

我们不能将错误类型的值赋给一个变量：

```
string name2 = 39;           // 错误：39 不是字符串
int number_of_steps = "Annemarie"; // 错误："Annemarie" 不是整数
```

编译器将会记录每个变量的类型，并确保对它的使用与定义它时的类型一致。

C++ 提供了相当多的类型（见附录 A.8）。但是，你只使用其中五种类型，就完全可以写出很好的程序：

```
int number_of_steps = 39;      // int 表示整数
double flying_time = 3.5;     // double 表示浮点数
char decimal_point = '.';      // char 表示单个字符
string name = "Annemarie";    // string 表示字符串
bool tap_on = true;           // bool 表示逻辑变量
```

命名为 `double` 是历史造成的：`double` 是“双精度浮点”的简写。浮点是数学概念上的一个实数在计算机中的近似值。

注意，这些类型中的每种都有自己文字（字面值）的特殊风格：

39	// int: 一个整数
3.5	// double: 一个浮点数
'.'	// char: 单引号限定的单个字符
"Annemarie"	// string: 双引号限定的字符串序列
true	// bool: 真或假

一串数字（例如 1234、2 或 976）表示一个整数，在单引号中的一个字符（例如 '1'、'@' 或 'x'）表示一个字符，具有小数点的一串数字（例如 1.234、0.12 或 .98）表示一个浮点值，在双引号中的一串字符（例如 "1234"、"Howdy!" 或 "Annemarie"）表示一个字符串。如果要获得字面值的详细描述，见附录 A.2。

3.3 输入和类型

输入操作 `>>` (“get from”) 是对类型敏感的，它读取的值与变量类型需要一致。例如： 

```
// 读取名字和年龄
int main()
{
    cout << "Please enter your first name and age\n";
    string first_name;    // 字符串变量
    int age;              // 整型变量
    cin >> first_name;   // 读入一个字符串
```

```

    cin >> age;           // 读入一个整数
    cout << "Hello, " << first_name << " (age " << age << ")\n";
}

```

因此，如果你输入 Carlos 22，>> 操作符将 Carlos 读入 first_name，将 22 读入 age，并且生成以下输出：

Hello, Carlos (age 22)

为什么它不将 Carlos 22（全部）读入 first_name？这是由于按照规定，读取字符串会被空白符所终止，包括空格、换行和 tab 字符。除此以外，空格在缺省情况下会被>> 忽略。例如，你可以在读取的数字之前添加任意多的空格；>> 将会跳过它们并读取这个数字。

如果你输入 22 Carlos，你将看到感到奇怪的东西，直到你能够理解这一切。22 将会读入 first_name，这是由于 22 毕竟是一个字符串。另一方面，Carlos 并不是一个整数，因此它不会被读取。这时的输出将是 22 和 age 再加上某个随机数，例如 -96739 或 0。为什么？你没有给 age 赋一个初始值，并且你没能成功地读入一个值存入它。因此，当你开始执行时，就会得到内存中的某个部分的“垃圾值”。在 10.6 节中，我们讨论“输入格式错误”的处理方式。现在，我们只是初始化 age，这样在输入错误时，我们会获得一个可预测的值：

```

// 读取名字和年龄（第 2 版）
int main()
{
    cout << "Please enter your first name and age\n";
    string first_name = "???"; // 字符串变量
    // ("???" 表示“不知道名字”)
    int age = -1; // 整型变量 (-1 表示“不知道年龄”)
    cin >> first_name >> age; // 读取字符串和整数
    cout << "Hello, " << first_name << " (age " << age << ")\n";
}

```

现在，输入 22 Carlos 将会输出：

Hello, 22 (age -1)

注意，我们可以在一个输入语句中读取几个值，就像我们可以在一个输出语句中写入几个值一样。注意，<< 和 >> 都对类型是敏感的，因此我们可以输出 int 型变量 age、string 型变量 first_name 以及字符串字面值 "Hello,"、"(age" 和 ")\n"。

使用 >> 的字符串读取（缺省）会被空格所终止；也就是它只能读取一个单词。但是，我们有时需要读取多个单词。当然会有多种方法来解决这个问题。例如，我们可以像这样来读取一个包括两个单词的名字：

```

int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second; // 读取两个字符串
    cout << "Hello, " << first << " " << second << '\n';
}

```

我们简单地使用 >> 两次，每次针对一个名字。当我们想输出多个名字时，必须在它们之间插入一个空格。

试一试

运行这个“名字和年龄”的例子。然后，修改它以月份的形式输出年龄：读取输入的年龄并（使用 * 操作符）乘以 12。将年龄读入一个 **double** 型的变量以便儿童使用（通常一个儿童会非常骄傲于自己已经 5 岁半了，而不是 5 岁）。

3.4 运算和运算符

除了指定什么值可以存储在一个变量中之外，变量类型决定我们可以对它进行什么运算和它们意味着什么。例如：

```
int count;
cin >> count;           // >> 读取一个整数到 count
string name;
cin >> name;           // >> 读取一个字符串到 name

int c2 = count+2;       // + 整数相加
string s2 = name + " Jr. "; // + 追加字符串

int c3 = count-2;       // - 整数相减
string s3 = name - " Jr. "; // 错误：对字符串来说，- 运算符是未定义的
```

通过“错误”，我们认识到编译器拒绝程序对字符串进行减法运算。编译器确切知道哪些运算可以应用于每个变量，这样可以防止很多错误的发生。但是，编译器不知道哪些值哪些运算对你有意义，因此它很高兴接受合法的运算，即使它们在你看来可能是荒谬的。例如：

```
int age = -100;
```

很明显，你不能有一个负的年龄（为什么不能？），但是没有人会告诉编译器，因此它会为这个定义生成代码。

这个表给出了一些常见和有用的类型可以使用的运算符：

	bool	char	int	double	string
赋值	=	=	=	=	=
加			+	+	
连接					+
减			-	-	
乘			*	*	
除			/	/	
余数 (模)			%		
递加 1			++	++	
递减 1			--	--	
加 n			+ = n	+ = n	
添加到结尾					+ =
减 n			- = n	- = n	
乘并赋值			* =	* =	
除并赋值			/ =	/ =	
余数并赋值			% =		

(续)

	bool	char	int	double	string
从 s 读到 x	s>>x	s>>x	s>>x	s>>x	s>>x
从 x 写到 s	s<<x	s<<x	s<<x	s<<x	s<<x
等于	==	==	==	==	==
不等于	!=	!=	!=	!=	!=
大于	>	>	>	>	>
大于或等于	>=	>=	>=	>=	>=
小于	<	<	<	<	<
小于或等于	<=	<=	<=	<=	<=

表中的空格表示一个运算符不能直接用于一种类型（尽管可能有间接使用这种运算符的方式，见 3.9.1 节）。我们将在后面的内容中解释这些及更多的运算符。这里的关键点是有很多有用的运算符，而且它们表示的意义对相似的类型通常是一样的。

我们来介绍一个涉及浮点数的例子：

```
// 练习运算符的简单程序
int main()
{
    cout << "Please enter a floating-point value: ";
    double n;
    cin >> n;
    cout << "n == " << n
        << "\nn+1 == " << n+1
        << "\nthree times n == " << 3*n
        << "\ntwice n == " << n+n
        << "\nn squared == " << n*n
        << "\nhalf of n == " << n/2
        << "\nsquare root of n == " << sqrt(n)
        << '\n'; // 在输出时表示新行 ("一行的结束")
}
```

很明显，常见的数学运算有常见的表示法和含义，这点和我们在小学中学到的知识一样。很自然，并不是我们想对一个浮点数做的任何事（例如得到它的平方根）都提供了对应的运算符。很多操作是通过命名函数来表示的。在这种情况下，我们使用标准库中的 `sqrt()` 来得到 `n` 的平方根 `sqrt(n)`。这种表示法与数学中相似。我们将会逐渐学习使用函数，并在 4.5 节和 8.5 节中讨论它们的细节。

试一试

运行这个小程序。然后，修改它以读取一个 `int` 型，而不是一个 `double` 型。注意，`sqrt()` 不是针对整数定义的，因此将 `n` 赋值为 `double` 型并对其执行 `sqrt()`。另外，“练习”一些其他操作。注意，对于 `int` 型来说，`/` 是整数除法，`%` 是余数（模），因此 $5/2$ 等于 2（而不是 2.5 或 3）， $5\%2$ 等于 1。整数 `*`、`/` 和 `%` 的定义，保证了两个正整数 `a` 和 `b` 可以得到 $a/b * a + a \% b == a$ 。

字符串拥有更少的运算符，不过在第 23 章中将看到它们有很多命名操作。但是，这些运算符的使用都符合常规。例如：

```
// 读取两个名字
int main()
{
    cout << "Please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second;           // 读入两个字符串
    string name = first + ' ' + second; // 字符串连接
    cout << "Hello, " << name << '\n';
}
```

字符串 + 意味着连接。也就是说，当 s1 和 s2 是字符串时，s1+s2 也是字符串，包含来自 s1 的多个字符跟着来自 s2 的多个字符。例如，如果 s1 的值为 "Hello"，s2 的值为 "World"，那么 s1+s2 的值为 "HelloWorld"。字符串比较也特别有用：

```
// 读入并比较两个名字
int main()
{
    cout << "Please enter two names\n";
    string first;
    string second;
    cin >> first >> second;           // 读入两个字符串
    if (first == second) cout << "that's the same name twice\n";
    if (first < second)
        cout << first << " is alphabetically before " << second << '\n';
    if (first > second)
        cout << first << " is alphabetically after " << second << '\n';
}
```

在这里，我们使用 if 语句来基于条件选择动作，该语句将在 4.4.1.1 节中详细介绍。

3.5 赋值和初始化

在很多方面，最有趣的运算符是赋值，表示为 =。它为一个变量赋予一个新的值。例如： 

int a = 3;	<i>// a 初始化为 3</i>	a: <table border="1" style="display: inline-table;"><tr><td>3</td></tr></table>	3	
3				
a = 4;	<i>// a 获得新值 4 (“变成 4”)</i>	a: <table border="1" style="display: inline-table;"><tr><td>4</td></tr></table>	4	
4				
int b = a;	<i>// b 的值为 a 的值的拷贝 (即为 4)</i>	a: <table border="1" style="display: inline-table;"><tr><td>4</td></tr></table> b: <table border="1" style="display: inline-table;"><tr><td>4</td></tr></table>	4	4
4				
4				
b = a+5;	<i>// b 获得新值 a+5 (即为 9)</i>	a: <table border="1" style="display: inline-table;"><tr><td>4</td></tr></table> b: <table border="1" style="display: inline-table;"><tr><td>9</td></tr></table>	4	9
4				
9				
a = a+7;	<i>// a 获得新值 a+7 (即为 11)</i>	a: <table border="1" style="display: inline-table;"><tr><td>11</td></tr></table> b: <table border="1" style="display: inline-table;"><tr><td>9</td></tr></table>	11	9
11				
9				

最后一次赋值需要注意。首先，很明显 = 并不意味着等于，a 不等于 a+7。它意味着赋 

值，也就是将一个新的值赋予一个变量。`a=a+7` 所做的事如下：

1. 首先，得到 `a` 的值，这里是整数 4。
2. 接着，将 7 和 4 相加，得到整数 11。
3. 最后，将整数 11 赋予 `a`。

我们也可以通过字符串来说明赋值：

```
string a = "alpha";      // a 初始化为 "alpha"
a: alpha

a = "beta";            // a 获得新值 "beta" (变成 "beta")
a: beta

string b = a;          // b 的值为 a 的值的拷贝 (即为 "beta")
a: beta
b: beta

b = a+"gamma";        // b 获得新值 a+"gamma" (即为 "betagamma")
a: beta
b: betagamma

a = a+"delta";        // a 获得新值 a+"delta" (即为 "betadelta")
a: betadelta
b: betagamma
```



以上，我们使用“初始化”和“获得新值”来区别两种相似、但是在逻辑上有区别的操作：

- 初始化（给一个变量它的初值）。
- 赋值（给一个变量一个新的值）。

这些运算是如此相似，因此 C++ 允许我们对它们使用相同的符号 (=)：

```
int y = 8;           // 将 y 初始化为 8
x = 9;              // 将 9 赋值给 x

string t = "howdy!"; // 将 t 初始化为 "howdy!"
s = "G'day";         // 将 "G'day" 赋值给 s
```

但是，赋值和初始化在逻辑上是不同的。你可以通过类型说明 (`int` 或 `string`) 来区分它们，它们总是从初始化开始；赋值并不需要这样做。从原则上来说，初始化时变量为空。另一方面，赋值在放入一个新的值之前，首先必须将旧的值清空。你可以将变量看作是一种小的盒子，值是一个可以放入其中的具体东西（例如一枚硬币）。在初始化之前盒子是空的，但是在初始化之后它总是包含一枚硬币，以便在里面放入一枚新的硬币。你（赋值操作符）首先需要移走旧的东西（“销毁旧的值”），你不能使盒子是空的。在计算机内存中并不完全如字面上所说，但是它对于我们理解后面的内容没有坏处。

3.5.1 实例：检测重复单词

当我们想将一个新的值放进一个对象，赋值是必需的。当你在考虑这件事情时，很明显

赋值在你做多次某件事情时是最有用的。当我们想以一个不同的值做事时，我们需要进行一次赋值。让我们来看一个小的程序，它在一连串单词中找到相邻的重复单词。这种代码是大多数的语法检查程序的一部分：

```
int main()
{
    string previous = " "; // 前一个单词，初始化为“不是一个单词”
    string current; // 当前单词
    while (cin>>current) { // 读入单词流
        if (previous == current) // 检测当前单词是否和前一个相同
            cout << "repeated word: " << current << '\n';
        previous = current;
    }
}
```

这个程序对我们并不是很有帮助，因为它没有告诉我们重复单词在文本中的哪个位置出现，但是现在它将会这样做。我们看从以下行开始的程序行：

```
string current; // 当前单词
```

这是一个字符串变量，我们使用它来读取当前（最近读入）的单词：

```
while (cin>>current)
```

这个结构称为一个 `while` 语句，它本身就很有意思，我们将在 4.4.2.1 节中详细介绍。`while` 的意思是：当输入操作 `cin>>current` 成功的情况下，`(cin>>current)` 后面的语句将反复执行，而 `cin>>current` 成功的条件是有字符串从标准输入中读取。注意，对于一个 `string`, `>>` 读取的是用空格分开的单词。你可以通过给程序一个终止输入符号（通常是指文件结尾）来终止这个循环。在 Windows 系统的计算机中，使用 `Ctrl+Z`（同时按 `Control` 和 `Z`）紧接着一个回车。在 Unix 或 Linux 系统的计算机中，使用 `Ctrl+D`（同时按 `Control` 和 `D`）。

因此，我们所做的是读取一个单词到 `current`，然后将它与前一个单词（存储在 `previous` 中）比较。如果它们是相同的，我们将会：

```
if (previous == current) // 检测当前单词是否和前一个相同
    cout << "repeated word: " << current << '\n';
```

然后，我们准备好对下一个单词重复进行上述操作。我们通过将 `current` 单词拷贝到 `previous` 中来进行这个操作：

```
previous = current;
```

这可以处理我们开始后的所有情况。但第一个单词没有前一个单词可以比较，代码该如何处理呢？这个问题可以在定义 `previous` 时得到解决：

```
string previous = " "; // 前一个单词，初始化为“不是一个单词”
```

这里 `" "` 只包含一个字符（空格字符，通过按键盘中的空格键来得到）。输入操作符 `>>` 会跳过空格，我们不可能通过输入得到它。因此，第一次执行 `while` 语句时，检测

```
if (previous == current)
```

失败（正如我们所希望的）。

理解程序流程的一种方式是“扮演计算机”，也就是按照代码逐行手工模拟程序执行。在一张纸上画出很多方块，然后在里面写入它们的值。按程序指定的方式修改储存在其中的值。



试一试

你亲自用一张纸来执行这个程序。输入是“**The cat cat jumped**”。对那些不十分明显的小段代码，即使是有经验的程序员也会用这种技术来可视化它们的执行。



试一试

运行“重复单词检测程序”。用句子“**She she laughed He He He because what he did did not look very very good good**”来测试它。这里有多少个重复的单词？为什么？单词在这里的定义是什么？重复单词的定义是什么？（例如，“**She she**”是否算是重复单词？）

3.6 复合赋值运算符

一个变量的递加（增加 1）在程序中很常用，C++ 为它提供了一个特定的语法。例如：

++counter

意味着

counter = counter + 1

有很多其他的常用方式，可以基于变量的当前值来修改它。例如，我们可能想将它加 7、减 9 或乘 2。C++ 直接支持这些运算。例如：

```
a += 7; // 表示 a = a+7
b -= 9; // 表示 b = b-9
c *= 2; // 表示 c = c*2
```

通常，对很多二元运算符 **oper**，**a oper= b** 意味着 **a = a oper b**（附录 A.5）。对于初学者，只需掌握运算符 **+=**、**-=**、***=**、**/=** 和 **%=**。这样提供了一种非常好的、紧凑的表示方法，可直接反映我们的意图。例如，在很多应用领域中，***=** 和 **/=** 被认为是“缩放”。

3.6.1 实例：重复单词计数

考虑上面的检测重复的相邻单词的例子。我们可以通过得到重复的单词在序列中的位置来改进程序。上述想法的一个简单变种是，我们可以统计单词数并输出重复单词数：

```
int main()
{
    int number_of_words = 0;
    string previous = " "; // 不是一个单词
    string current;
    while (cin >> current) {
        ++number_of_words; // 增加单词数
        if (previous == current)
            cout << "word number " << number_of_words
            << " repeated: " << current << '\n';
        previous = current;
    }
}
```

我们将单词计数器初始化为 0。我们每次读入一个单词，就会将这个计数器递增：

```
++number_of_words;
```

因此，第一个单词变为数值 1，下一个单词变为数值 2，等等。我们也可以按以下方式完成相同功能：

```
number_of_words += 1;
```

或者是

```
number_of_words = number_of_words + 1;
```

但是，`++number_of_words` 更加简短，并且直接表达递增的思想。

注意，这个程序与 3.5.1 节中的一个程序是如此相似。很明显，我们只是将这个程序从 3.5.1 节拿来，并对它进行一点儿修改以实现我们的目标。这是我们解决一个问题时很常用的技术，寻找一个相似的问题并用我们的方案加以修改。不要从零开始，除非你不得不这样  做。在一个程序前期版本的基础上修改通常会节省大量时间，原始程序中的一切努力成果都将为我们所用。

3.7 命名

我们命名自己的变量，这样我们可以记住它们，并在程序的其他部分中使用。在 C++ 中什么可以是一个名字呢？在一个 C++ 程序中，一个名字必须以一个字母开始，并且只能包含字母、数字和下划线。例如：

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

以下不是名字：

```
2x           // 名字必须以字母开头
time$to$market // $ 不是字母、数字或下划线
Start menu    // 空格不是字母、数字或下划线
```

当我们说“不是名字”时，我们的意思是 C++ 编译器不认为它们是名字。

如果你阅读系统代码或机器生成的代码，你可能看到以下划线开始的名字，例如 `_foo`。 你自己不要这样写，这样的名字是为实现和系统实体保留的。尽量避免使用下划线，这样你将不会看到你的名字与实现生成的名字冲突。

名字是区分大小写的；也就是说，大写字母和小写字母是不同的，因此 `x` 和 `X` 是不同的名字。这个小程序至少有 4 个错误：

```
#include "std_lib_facilities.h"

int Main()
{
    STRING s = "Goodbye, cruel world!";
    cOut << s << '\n';
}
```

在定义名字时用大小写来区分，例如 `one` 和 `One`，通常不是一个好主意；它不会使编译器混淆，但是它会使程序员混淆。



试一试

编译“Goodbye, cruel world!”程序，并且检查错误信息。编译器是否能发现所有错误？它对出现问题的建议是什么？编译器是否混淆并发现超过4个错误？按出现顺序依次改正这些错误，看错误信息如何变化（和改进）。

C++语言保留了很多（大约85个）名字作为“关键字”。我们将在附录A.3.1中列出它们。你不能使用它们作为变量、类型、函数等的名字。例如：

```
int if = 7; // 错误：if是关键字
```

你可以使用标准库中的内容（例如**string**）作为名字，但是你不应该这样做。如果你想要使用标准库的话，这样一个通用名字的重用将会带来麻烦：

```
int string = 7; // 这会带来麻烦
```

当你为自己的变量、函数、类型等选择名字时，最好选择有特定含义的名字；也就是说，选择有助于人们理解你的程序的名字。如果你将变量胡乱命名为“简单型”的名字，例如x1、x2、s3与p7，则你在理解程序要做什么时也会遇到问题。缩写和仅有首字母的缩写会使人糊涂，因此要谨慎使用它们。下面这些仅有首字母的缩写对我们很明显，但是我们认为你将对至少一个有疑问：

```
mtbf  
TLA  
myw  
NBV
```

若几个月之后我们再来看这些，同样会对至少一个有疑问。

短名字（例如*x*与*i*）在常规使用时是有含义的；也就是说，*x*可能是一个本地变量或一个参数（见4.5节与8.4节），*i*可能是一个循环的索引（见4.4.2.3节）。

不要使用很长的名字；它们难以输入，由于太长难以在一个屏幕中显示，也难以快速阅读。下面的名字可能是合适的：

```
partial_sum  
element_count  
stable_partition
```

这些名字可能太长：

```
the_number_of_elements  
remaining_free_slots_in_symbol_table
```

我们的风格是在一个标识符中使用下划线来区分单词（例如**element_count**），而不是其他可选方案（例如**elementCount**与**ElementCount**）。我们不使用全部大写字母的名字（例如**ALL_CAPITAL LETTERS**），这是由于它们通常保留作为宏（27.8节与附录A.17.2），因此我们避免这样使用。我们使用首字母大写来定义自己的类型，例如**Square**与**Graph**。C++语言和标准库不使用大写字母，因此它们使用**int**而不是**Int**，使用**string**而不是**String**。因此，我们的规定会帮助你减少对自己类型和标准类型的混淆。



避免使用容易输错、误读或混淆的名字。例如：

Name	names	nameS
foo	foo	fl
f1	fl	fi

字符 0 (数字零)、o (小写字母 o)、O (大写字母 O)、1 (数字 1) 与 l (小写字母 l)，特别是 l 很容易引起麻烦。

3.8 类型和对象

类型是 C++ 和大多数编程语言的核心内容。接下来我们以更技术性的观点更近距离地讨论类型，特别是我们在计算过程中用来存储数据的对象类型。长远来看这会节省你的时间，它也可以避免引起你的混淆。

- 类型定义一组可能的值与一组运算（对于一个对象）。
- 对象是用来保存一个指定类型值的一些内存单元。
- 值是根据一个类型来解释的内存中的一组比特。
- 变量是一个命名的对象。
- 声明是命名一个对象的一条语句。
- 定义是一个声明，但同时也为对象分配了内存空间。

通俗地说，我们可以将一个对象看作一个盒子，我们可以将指定类型的值放入它。一个 int 盒子可以保存整数，例如 7、42 与 -399。一个 string 盒子可以保存字符串值，例如 “Interoperability”、“tokens: !@#\$%^&*” 与 “Old MacDonald had a farm”。我们可以用图表来考虑：

<code>int a = 7;</code>	a:	7
<code>int b = 9;</code>	b:	9
<code>char c = 'a';</code>	c:	a
<code>double x = 1.2;</code>	x:	1.2
<code>string s1 = "Hello, World!";</code>	s1:	13 Hello, World!
<code>string s2 = "1.2";</code>	s2:	3 1.2

由于 string 要跟踪它保存的字符数，因此 string 比 int 的表示方法更复杂。注意，一个 double 保存一个数字，而一个 string 保存的是字符序列。例如，x 保存数字 1.2，而 s2 保存三个字符 '1'、'.' 与 '2'。字符和字符串常量的引号并不保存。

每个 int 的大小是相同的；也就是说，编译器为每个 int 分配相同的固定大小的内存。在一个典型的台式计算机中，这个大小是 4 个字节 (32 个比特)。与此类似，bool、char 与 double 是固定大小的。在通常情况下，你会发现台式计算机为一个 bool 或 char 分配 1 个字节 (8 个比特)，为一个 double 分配 8 个字节。注意，不同类型的对象使用不同大小的空间。特别地，一个 char 比一个 int 占用更少的空间，string 不同于 double、int 与 char，不同大小的字符串占用不同大小的空间。

在内存中比特的含义完全依赖于访问它所用的类型。我们这样思考此问题：计算机内存不知道我们的类型，只是将它保存起来。只有当我们决定内存如何解释时，在内存中的比特才有意义。这个过程与我们每天使用数字相似。12.5 的含义是什么？我们并不知道。它可能是 12.5 美元、12.5 厘米或 12.5 加仑。只有当我们提供一个单位时，12.5 才有意义。

例如，同样的内存比特，当表示一个 int 时为 120，而表示一个 char 时为 'x'。如果我们将它看成一个 string，它将不会有意义，并在我们试图使用它时出现运行时错误。我们可以

如下以图的形式来解释，其中使用 1 和 0 表示内存中的比特值：

00000000 00000000 00000000 01111000

这是一个内存区域（一个字）中的一组比特，它们可以被读取为一个 int (120) 或一个 char ('x'，只看最右侧的 8 个比特）。一个比特是计算机中的一个内存单元，它可以保存一个值 0 或 1。想理解二进制数的含义，见附录 A.2.1.1。

3.9 类型安全

每个对象在定义时被分配一个类型。对于一个程序或程序的一个部分，如果使用的对象符合它们规定的类型，那么它们是类型安全的。不幸的是，有些执行的操作是类型不安全的。例如，在一个变量没有初始化之前使用它，被认为是类型不安全的：

```
int main()
{
    double x;           // 此处忘记初始化;
                        // x 的值是未定义的
    double y = x;       // y 的值也是未定义的
    double z = 2.0+x;   // + 的意义和 z 都是未定义的
}
```

当没有初始化的 x 被使用时，有的实现甚至可以给出一个硬件错误。永远记住初始化你的变量！这个规则的例外非常少，例如我们立即将一个变量作为输入操作的目标，但是记住初始化变量是一个好习惯，它会为我们减少很多的麻烦。

完全的类型安全是理想的，因此它是语言的一般性规则。不幸的是，C++ 编译器不能保证完全的类型安全，但是通过良好的代码训练和运行检查，我们可以避免违反类型安全。理想情况是绝不使用编译器不能保证类型安全的语言特性：静态类型安全。不幸的是，它对于大多数有趣的编程应用过于严格。很明显低效的做法是，编译器隐式地生成代码来检查是否违反类型安全并全部标记它们，C++ 并不支持这种方式。当我们决定做（类型）不安全的事时，我们必须自己做某些检查工作。当在本书中遇到这种情况时，我们将会指明。

类型安全的思想在编写代码时非常重要。这是我们在这本书的靠前章节花费时间介绍它的原因。请注意陷阱并避开它们。

3.9.1 安全转换

在 3.4 节中，我们发现不能直接对 char 进行相加，或者将一个 double 与一个 int 比较。但是，C++ 提供了间接方式来完成这些操作。在有必要时，一个 char 可以转换成一个 int，而一个 int 也可以转换成一个 double。例如：

```
char c = 'x';
int i1 = c;
int i2 = 'x';
```

这里的 i1 和 i2 都被赋值为 120，它是字符 'x' 在最流行的 8 比特字符集 ASCII 中的整型值。这是一个简单和安全的方法，通过它可以获得一个字符的数字表示。我们称这种 char-int 的转换为安全的，这是由于没有信息丢失；也就是说，我们可以将 int 结果拷贝回一个 char 中，并且得到原始的值：

```
char c2 = i1;
cout << c << ' << i1 << ' << c2 << '\n';
```

输出结果为

x 120 x

一个值被转换成一个等价的值，或是一个最接近等价的值（对于 `double`），在这种意义下，下面这些转换就是安全的：

```
bool 到 char
bool 到 int
bool 到 double
char 到 int
char 到 double
int 到 double
```

最有用处的转换是从 `int` 到 `double`，这是由于它允许在表达式中混合使用 `int` 和 `double`：

```
double d1 = 2.3;
double d2 = d1+2;      // 相加之前，2 转换为 2.0
if (d1 < 0)            // 比较之前，0 转换为 0.0
    cout << "d1 is negative";
```

对于一个确实很大的整数，当它被转换成 `double` 时，我们（在有些计算机中）会有一些精度上的损失。这是一个不常见的问题。

3.9.2 不安全转换

安全的转换对程序员通常是一个福音，它可以简化代码编写。不幸的是，C++ 也允许  (隐式的) 不安全转换。所谓的不安全，我们的意思是一个值可以转换成一个其他类型的值，这个值不等于原始的值。例如：

```
int main()
{
    int a = 20000;
    char c = a;      // 尝试将一个大整数“压缩”进小的字符型
    int b = c;
    if (a != b)      // != 表示“不等于”
        cout << "oops!: " << a << "!=" << b << '\n';
    else
        cout << "Wow! We have large characters\n";
}
```

这种转换又被称为“窄化”转换，这是由于它们将一个值放入一个对象，这个对象可能太小（“狭窄”）以至于不能存放这个值。不幸的是，只有少数编译器会警告将 `char` 初始化为 `int` 的不安全。这里的问题是一个 `int` 通常比一个 `char` 大，因此（在这种情况下）它可以保存一个并不能表示 `char` 的 `int` 值。尝试执行这个程序，查看你计算机中的值 `b`（常见的结果是 32）；更进一步，完成实验：

```
int main()
{
    double d = 0;
    while (cin>>d) {           // 重复执行下面的语句
        // 只要我们不断输入数
```

```

int i = d;           // 尝试压缩 double 型到 int 型
char c = i;          // 尝试压缩 int 型到 char 型
int i2 = c;          // 获取该字符的整型值
cout << "d==" << d           // 原始的 double 值
    << " i==" << i           // 转换成的 int 值
    << " i2==" << i2         // 字符的 int 值
    << " char(" << c << ")\\n"; // 字符值
}
}

```

我们使用 `while` 语句允许尝试很多值，这个语句将在 4.4.2.1 节中解释。

试一试

输入各种各样的值来运行这个程序。尝试小的值（例如 2 和 3）；尝试大的值（大于 127、大于 1000）；尝试负值；尝试 56；尝试 89；尝试 128；尝试非整型值（例如 56.9 和 56.2）。除了展示在你的机器上如何从 `double` 转换成 `int`，以及如何从 `int` 转换成 `char`，本程序还显示了对给定整型值，你的机器会输出哪个字符（如果有的话）。

你将发现很多输入值产生“不合理”的结果。基本上，我们是在尝试将一加仑水倒入容量为一品脱的器皿中（大约是将 4 升水倒入一个 500 毫升的杯子）。下面所有的转换：

```

double 到 int
double 到 char
double 到 bool
int 到 char
int 到 bool
char 到 bool

```

都会被编译器接受，即便它们是不安全的。我们说的不安全是指它们保存的值可能与赋的值不同。为什么这会是一个问题？这是由于我们通常不去怀疑发生了一个不安全的转换。考虑：

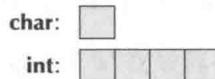
```

double x = 2.7;
// 很多代码
int y = x;      // y 变成了 2

```

在我们定义 `y` 时可能忘记 `x` 是一个 `double`，或者我们临时忘记 `double` 到 `int` 转换会截断（总是去掉小数点后的尾数），而不是使用常用的四舍五入。发生的事情是完全可以预测的，但是在 `int y=x;` 处没有任何能提醒我们信息 (.7) 被丢掉。

从 `int` 到 `char` 的转换不会出现截断的问题（`int` 和 `char` 都不能表示小数部分）。但是，一个 `char` 只能保存非常小的整型值。在一台 PC 机中，一个 `char` 占用 1 个字节，而一个 `int` 占用 4 个字节。



 因此，我们不能将一个大的数（例如 1000）放入一个 `int` 而不丢失任何信息：这个值是“窄化”的。例如：

```
int a = 1000;  
char b = a;      // b 为 -24 (在一些机器上)
```

不是所有的 int 值都有等价的 char, char 值的确切范围依赖于特定的实现。在一台 PC 机中, char 值的范围是 [-128:127], 但是为了可移植性, 只有 [0:127] 可以使用。这是由于并不是每台计算机都是 PC 机, 不同计算机的 char 值的范围不同, 例如有的是 [0:255]。

为什么人们能接受窄化转换引起的问题? 主要原因是历史性的: C++ 从它的前辈语言 C 继承了窄化转换, 因此从 C++ 出现时很多代码就依赖于窄化转换。很多这种转换实际上不会引起问题, 这是由于它们所涉及的值碰巧在范围内, 并且很多程序员反对编译器“告诉它们做什么”。特别是对有经验的程序员来说, 这些不安全转换问题在小的程序中是可管理的。但它们在很大的程序中经常导致错误, 也是新手程序员遇到问题的关键所在。不过, 很多编译器可以对窄化转换发出警告。

C++11 引入了一种初始化方式, 可彻底避免窄化转换。例如, 我们可以(也应该) 使用 `{}` 列表记号来重写上面的问题代码, 而不是用 `=` 记号:

```
double x {2.7};    // 正确  
int y {x};         // 错误: double → int 可能窄化  
  
int a {1000};      // 正确  
char b {a};        // 错误: int → char 可能窄化
```

当初始化值是整数字面值时, 编译器会检查实际的值并接受那些不会引起窄化转换的值:

```
int char b1 {1000}; // 错误: 窄化 (假设 8 比特字符型)  
char b2 {48};       // 正确
```

如果你认为转换可能导致一个错误值, 那么你需要做什么? 使用 `{}` 初始化来避免意外, 并且当需要做转换时, 像本节中的第一个例子那样在赋值之前先对值进行检查。5.6.4 节与 7.5 节介绍了做这种检查的简单方式。`{}` 列表记号也被称为通用统一初始化, 我们以后会看到它的更多使用。

简单练习

在完成这个练习的所有步骤之后, 运行你的程序以确认它确实完成你希望它做的事。列出那些曾经出现的错误, 这样以后可以尽量避免它们。

1. 这个练习是编写一个程序, 基于用户输入生成一封简单格式的信。首先, 输入来自 3.1 节的代码, 提示用户输入他或她的名字, 并且输出 “Hello, `first_name`”, 这里的 `first_name` 是用户输入的名字。然后, 按以下要求修改你的代码: 将提示修改为 “Enter the name of the person you want to write to”, 并将输出修改为 “Dear `first_name`,”。不要忘记逗号。
2. 增加一行或两行前言, 例如 “How are you? I am fine. I miss you.” 确定首行需要缩进。增加由你选择的几行内容, 这毕竟是你的信。
3. 现在, 提示用户输入另一个朋友的名字, 将它保存在 `friend_name` 中。在你的信中增加一行: “Have you seen `first_name` lately?”
4. 声明一个 `char` 变量为 `friend_sex`, 并将它的值初始化为 0。如果这个朋友是男性, 提示用户输入一个 `m`; 如果这个朋友是女性, 提示用户输入一个 `f`。将该值赋给变量 `friend_sex`。然后, 使用两个 `if` 语句完成以下输出:

- 如果这个朋友是男性，输出 “If you see friend_name please ask him to call me.”。
- 如果这个朋友是女性，输出 “If you see friend_name please ask her to call me.”。
5. 提示用户输入收信人的年龄，并为它分配一个 int 变量 age。让你的程序输出 “I hear you just had a birthday and you are age years old.” 如果 age 小于等于 0 或大于等于 110，调用 simple_error (“you're kidding!”)，其中 simple_error 包含在 std_lib_facilities.h 中。
6. 在你的信中增加：
- 如果你朋友的年龄小于 12，输出 “Next year you will be age+1.”。
- 如果你朋友的年龄等于 17，输出 “Next year you will be able to vote.”。
- 如果你朋友的年龄大于 70，输出 “I hope you are enjoying retirement.”。
- 运行程序，确保对不同的值都输出正确。
7. 增加 “Yours sincerely,” 接着是两个空行用于签名，再接着是你的名字。

思考题

1. 术语 prompt 的含义是什么？
2. 哪种操作符用于读入值到一个变量？
3. 如果你希望用户在你的程序中为一个命名为 number 的变量输入一个整型值，如何用两行代码来提示用户输入并将值输入你的程序中？
4. \n 的名称是什么，它的目的是什么？
5. 怎样终止输入一个字符串？
6. 怎样终止输入一个整数？
7. 如何将


```
cout << "Hello, ";
cout << first_name;
cout << "\n";
```

 编写为一行代码？
8. 什么是对象？
9. 什么是字面值常量？
10. 有哪些不同类型的字面值常量？
11. 什么是变量？
12. char、int 和 double 的典型大小是多少？
13. 我们用哪种方式测试内存中的实体（例如 int 和 string）大小？
14. 操作符 = 与 == 之间的区别是什么？
15. 什么是一个定义？
16. 什么是初始化，它和赋值的区别是什么？
17. 什么是字符串连接，如何使它在 C++ 中工作？
18. 在以下名字中，哪些在 C++ 中是合法的？如果一个名字是不合法的，为什么？

This_little_pig	This_1_is_fine	2_For_1_special
latest thing	the_\$12_method	_this_is_ok
MiniMineMine	number	correct?

19. 请举出 5 个你不会使用的合法名字的例子，因为它们容易引起混淆。
 20. 选择名字的好规则有哪些？

21. 什么是类型安全，为什么它是重要的？
22. 为什么从 `double` 转换成 `int` 是一件坏事？
23. 请定义一个判断从一种类型到另一种类型的转换是否安全的规则。

术语

assignment (赋值)	definition (定义)	operation (运算)
<code>cin</code>	increment (递增)	operator (运算符)
concatenation (连接)	initialization (初始化)	type (类型)
conversion (转换)	name (名字)	type safety (类型安全)
declaration (声明)	narrowing (窄化)	value (值)
decrement (递减)	object (对象)	variable (变量)

习题

1. 如果你还没有开始这样做，请先做本章的“试一试”练习。
2. 编写一个 C++ 程序，将英里转换成公里。你的程序应该有一个合理的提示，要求用户输入一个表示英里的数字。提示：1 英里等于 1.609 公里。
3. 编写一个程序，不做其他的任何事情，只声明一系列合法与不合法的变量名（例如 `int double=0;`），这样你可以看到编译器的反应。
4. 编写一个程序，提示用户输入两个整型值。将这些值保存在 `int` 变量 `val1` 和 `val2` 中。编写你的程序求这两个值中的最小值、最大值、和、差、乘积和比率，并且将结果输出给用户。
5. 修改上个程序，让用户输入浮点数值并将它们保存在 `double` 变量中。比较你选择的多种输入在两个程序的输出。这些结果是否相同？它们是否应该相同？区别是什么？
6. 编写一个程序，提示用户输入三个整型值，然后按数值大小顺序用逗号隔开的方式输出这些值。因此，如果用户输入值为 10 4 6，输出值应为 4,6,10。如果有两个值相同，那么将它们按序一起输出。因此，输入 4 5 4 将会输出 4,4,5。
7. 做习题 6，但是输入 3 个字符串。因此，如果用户输入“Steinbeck”、“Hemingway”和“Fitzgerald”，输出将是“Fitzgerald, Hemingway, Steinbeck”。
8. 编写一个程序，测试一个整型值是奇数还是偶数。确保你的输出是清楚和完整的。换句话说，不要只是输出“yes”或“no”。你的输出应该是独立的，例如“The value 4 is an even number。”提示：阅读 3.4 节中的余数（模）操作。
9. 编写一个程序，将拼写的数字（例如“zero”和“two”）转换成数值（例如 0 和 2）。当用户输入一个拼写的数字，程序将打印出对应的数字。针对数值 0、1、2、3 和 4 完成这个操作，如果用户输入没有对应的值（例如“stupid computer!”），程序输出“not a number I know”。
10. 编写一个程序，输入运算符和两个操作数，输出计算结果。例如：

+ 100 3.14
* 4 5

将操作符读入到一个称为 `operation` 的字符串，用一个 `if` 语句判断哪个操作是用户希望的，例如 `if (operation=="+")`。将操作数读入到 `double` 类型的变量。实现 +、-、*、/（很明显，分别代表加、减、乘、除）几种运算。

11. 编写一个程序，提示用户输入一些硬币的数量，包括 pennies (1 美分硬币)、nickels (5 美分硬币)、dimes (10 美分硬币)、quarters (25 美分硬币)、half dollars (50 美分硬币) 和 one-dollar (100 美分硬币)。分别询问用户每种面额的硬币数量，例如“How many pennies do you have?”（“你有几个 1 美分硬币”），程序将输出类似以下的内容：

```
You have 23 pennies.  
You have 17 nickels.  
You have 14 dimes.  
You have 7 quarters.  
You have 3 half dollars.  
The value of all of your coins is 573 cents.
```

改进你的程序：如果只有一枚硬币被报告，确认输出语法是正确的，例如“14 dimes”和“1 dime”（不是“1 dimes”）。另外，将总数用美元和美分来表示，例如用 \$5.73 来代替 573cents。

附言

请不要低估类型安全概念的重要性。类型是大多数正确程序的核心概念，大多数用于构建程序的有效技术依赖于类型的设计与使用，正如我们将在第 6 章、第 9 章以及后续章节中看到的一样。

计算

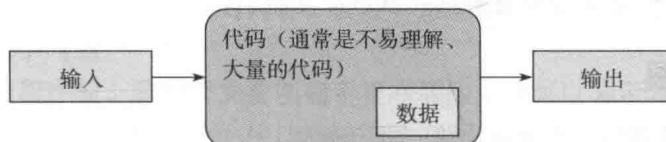
如果不要求结果的正确性，我可以让程序运行得任意快。

——Gerald M. Weinberg

本章将介绍一些与计算相关的基本概念。我们将着重讨论如何通过一系列指令来计算一个数值量（表达式，expression）；如何在可替代运算中进行选择（Selection）；如何对一系列数据进行重复计算（迭代，Iteration）。此外，我们还将介绍一种可以专门划分出来并命名的子计算（函数，Function）。本章内容的核心是通过介绍表达计算的良好方法，能形成正确而且规范的程序。为了让读者更好地理解计算，我们将引入 `vector`（向量）类型来表示数据序列。

4.1 简介

有一种观点认为，程序就是以计算为目的的，即程序都要有输入和输出。在这里，我们把能够运行程序的硬件设备称为计算机。如果我们用广义的概念来理解输入和输出的话，那么上述的观点可以认为是正确的。



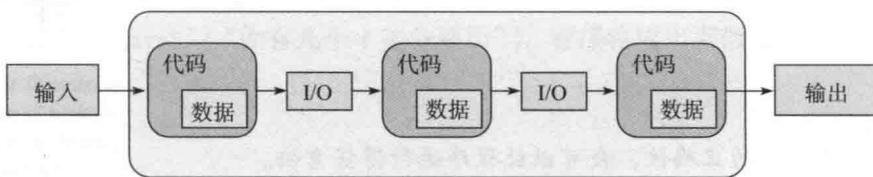
程序的输入来源很多：可以是键盘、鼠标、触摸屏、文件、其他输入设备、其他程序，或者同一程序的其他部分。在这里，“其他输入设备”的范围很广，它表示了一大类实际输入设备：音乐键盘、摄像机、网络设备、温度传感器、数字图像传感器等等。随着技术的进步，输入设备可以千变万化。

为了处理输入，程序通常包含一些数据，有时被称为其数据结构或状态。例如，日历程序需要记录不同国家的公共假期和用户的事务安排表。这些数据一部分是在程序中设定好的；还有一部分是在程序运行期间，程序通过各种输入设备获取的。例如，通过用户的输入，日历程序可以准确地建立用户的事务安排表。对于一个日历程序来说，主要输入包括对日期的查询（一般通过鼠标点击）和对用户事务安排表的处理（通常使用键盘输入相关信息）。输出包括日历和事务安排表的显示、程序按钮和提示符显示等。

输入的来源非常广泛。同样，输出也有很多不同的途径：可以是屏幕、文件以及其他设备，或者其他程序，甚至可以是同一程序的其他部分。输出设备很多，例如网卡、音乐合成器、电动马达、发光器和加热器等。

从编程的角度看，最重要也是最有趣的两类输入、输出是“从其他程序输入或输出”和“从同一程序的不同子程序输入或输出”。本书后续的大部分内容可以被视为后一种类型的

实例：在协作完成一个大的软件时，应该如何合理地设计程序结构，并能够保证每一个子程序之间都能够正确地共享和交换数据？这是编程的核心问题，下图说明了这一过程：



其中，I/O 是“input/output”的缩写。在上图中，一部分代码的输出是下一部分代码的输入。而子程序之间的数据共享可以通过内存、非易失存储设备（例如硬盘）或者网络完成。在这里，“子程序”是指程序中的各类函数，包括：根据输入参数产生输出结果的函数（例如求浮点数的平方根函数）；对物理对象进行操作的函数（例如在屏幕上画线的函数）；修改现有程序数据表的函数（例如在顾客信息表中增加一个姓名）。

通常意义上的“输入”和“输出”是指信息进入和离开计算机。正如前文所述，也可以将“输入”和“输出”引申到子程序的数据传递。通常，子程序的输入被称为参数，子程序的输出被称为结果。

从这个意义上来说，计算就是基于输入生成输出的过程。例如，基于参数 7（输入），利用计算过程（函数）square 可以得到结果 49（输出）（具体细节见 4.5 节）。有趣的是，直到 20 世纪 50 年代，计算机（computer，计算器）的定义还是一个从事计算任务的人，例如会计、导航员和物理学家等。今天，人类社会的大部分计算任务都是由各种类型的计算机（真正的机器）完成的。日常生活中最常见的就是计算器了。

4.2 目标和工具



程序员的任务就是将计算表达出来，并且做到：

- 正确；
- 简单；
- 高效。

请注意上述顺序。一个输出错误结果的快速程序是没有任何意义的。同样，一个正确、高效但是非常复杂的程序，最终的结果往往是被放弃或者重写。注意，为了适应不同的需求和硬件环境，有用的程序总会被无数次改写。因此，一个程序，或者它的任一子程序，应该以尽可能简单的方式来实现。举个例子，假设你为学校的孩子写了一个非常棒的算术教学程序，但是这个程序的内部组织非常糟糕。这个程序必须与孩子们交互。那它应该用什么语言呢？英语？又或是西班牙语？如果程序要在芬兰或科威特用，那又该怎么办呢？无疑，为了能适应不同地区的需要，程序使用的交互语言必须能够被修改。如果程序的结构非常混乱，那么原本逻辑上很简单的修改操作会变成一项艰巨的任务。



在我们开始编写代码的时候，就要特别关注正确、简单和高效这三个基本原则，这也是专业程序员的最重要职责。在实际工作中，遵循这些原则意味着代码仅仅能用是不够的，我们必须认真考虑代码的结构。一个看似矛盾的事实是，关注代码结构和“代码质量”是程序取得成功的最快途径。这一点很好理解：编程时对编码结构和质量所付出的努力，可以大大简化最令人沮丧的编程工作——调试。这是因为好的程序结构不但可以减少错误的发生，而

且还能缩短发现并改正错误的时间。

程序的组织体现了程序员的编程思路，目前的手段主要是把一个大的计算任务划分为许多小任务。这一技术主要包括两类方法：

- **抽象**：即不需要了解的程序具体实现细节被隐藏在相应的接口之后。例如，为了实现电话本的排序，我们不需要了解排序算法的细节（已经有很多书讨论如何排序了），我们要做的只是调用 C++ 标准库中的 `sort` 算法就可以了。关于排序，我们只需知道如何调用此算法，因此我们可以编写 `sort(b)`，其中 `b` 表示电话本；`sort()` 是标准库中 `sort` 算法（16.8 节，附录 C5.4）的一个变种（16.9 节），定义在 `std_library.h`。另一个例子是内存的使用，直接使用内存空间是一个糟糕的想法。通常，我们通过变量（3.2 节）、标准库 `vector`（4.6 节，12 ~ 14 章）和 `map`（16 章）等来访问内存。
- **分治**：即把一个大问题分为几个小问题分别解决。例如，在建立一个字典的时候，可以把这一任务分解为三个子任务：读数据、数据排序和输出数据，每个子任务都明显小于原来的任务。

这么做有什么用呢？毕竟，由很多部分构成的程序要比一个完整的程序规模大。直接原因是大问题处理起来太困难，这种情况不只存在于程序设计，也存在于很多其他领域。对于这一情况，我们的解决办法是：将问题不断分解、细化，直到问题小到能够被我们很好地理解和解决为止。以编程为例，一个 1000 行程序的规模是一个 100 行程序的 10 倍。但是 1000 行程序的错误会远超过 100 行程序的 10 倍。解决的办法就是把这个 1000 行程序分解为多个子程序，每个子程序不超过 100 行。对于大型软件来说，例如一个 1 千万行的软件，使用抽象和分治等技术就不只是一个编程建议了，而是必须这样做。编写并维护一个庞大的单一程序是很困难的。对于本书剩余的内容，读者不妨尝试利用已有的工具和方法，把一些大问题分解为一系列小问题。

在考虑划分一个程序前，我们首先要明确手里有哪些工具可以表示各个子程序及其之间的关系。这是因为一个能够提供充分的接口和功能的库可以大大简化程序的划分工作。凭空想象什么是程序的最优划分方法是不切实际的，按照功能对程序进行划分是目前最常用的方法。无疑，利用已有的各类程序库能够简化按功能进行程序划分的工作量。事实上，利用类似 C++ 标准库这种已有的库，不但可以减少编程的工作量，而且可以减少测试和写文档的工作量。例如，`iostreams` 库屏蔽了 I/O 设备的实现细节。程序员可以直接调用相应库函数，而不需要了解具体 I/O 接口是如何实现的，这就是抽象方法的一个具体实例。在后续章节中，我们将展示更多例子。

需要记住的是，仅通过编写大量语句是不会得到好代码的，更要注意程序的组织结构。为什么要在这里反复强调这点呢？因为到目前为止，大部分初学者对如何写代码还没有一个完整的认识。此时，强调程序的组织和结构有助于初学者建立良好的编程习惯。初学者在编程时经常犯的一个错误是：不是首先认真分析问题，确定程序的组织架构，而是急着去写程序，结果一头陷进一些技术细节里面，忘掉了更重要的程序结构问题。对于一个好的程序员和系统设计师，软件的结构问题是在开发过程中始终要关注的最重要问题。混乱的软件结构将导致代价高昂的软件维护和升级。打个比方来说，缺少组织的软件就像用土坯来盖房，虽然房子可以盖起来，但你不要指望能盖高楼，因为土坯是无法支撑楼房的重量的。如果你希望自己的软件有生命力，那么在一开始就应该注意软件的组织结构，而不是等到遇到挫折后再回过头来重建它。

4.3 表达式

 表达式是程序的最基本组成单元。表达式就是从一些操作数计算一个值。最简单的表达式是字面常量，例如 10, 'a', 3.14 和 "Norah"。

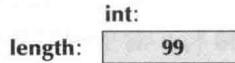
变量名也是一种表达式，变量表示与名字对应的那个对象。例如，

```
// 计算部分
int length = 20;           // 整型字面常量（用于变量的初始化）
int width = 40;
int area = length*width;   // 乘法操作
```

在这里，字面常量 20 和 40 用于初始化变量 `length` 和 `width`。然后，`length` 和 `width` 进行乘法操作，即 `length` 和 `width` 所表示的值相乘。这时，`length` 可以理解名字为 `length` 的变量的值。考虑如下情况：

```
length = 99; // 把 99 赋给 length
```

该语句中的 `length` 位于赋值号左边（即 `length` 是左值），其含义是名字为 `length` 的变量，因此赋值表达式的含义是“把 99 赋给名为 `length` 的变量”。要注意区分 `length` 用于赋值运算符左边和右边的含义是不同的，`length` 在左边时（即 `length` 是左值）表示“名为 `length` 的变量”，在右边时（即 `length` 是右值）表示“名为 `length` 的变量的值”。通过下面的图可以更清楚地解释这个概念：



上图表示了一个名为 `length` 的整型变量，其值为 99。当 `length` 是左值时，`length` 表示这个变量本身；当 `length` 是右值时，`length` 表示这个变量的值。

通过加、减、乘、除等运算符，我们还可以表示更复杂的表达式。如果需要的话，使用括号还可以构成复合表达式：

```
int perimeter = (length+width)*2; // 先加法后乘法
```

如果没有括号的话，表达式必须表示为：

```
int perimeter = length*2+width*2;
```

显然，这种表示方法很繁琐，而且容易出错：

```
int perimeter = length+width*2; // 先 width 乘 2 再和 length 相加
```

这个语句的错误是语义错误而不是语法错误，编译器认为是合法的，该语句通过一个有效的表达式初始化 `perimeter` 变量。由于编译器不清楚 `perimeter` 的数学定义，因此编译器不能确定表达式是否有意义。由此造成的表达式计算结果无意义，这显然是编程人员的问题。

按照运算符的优先级规则，`length+width*2` 表示 `length+(width*2)`，同样，`a*b+c/d` 表示 `(a*b)+(c/d)`，而不是 `a*(b+c)/d`。附录 A.5 给出了运算符优先级表。

使用括号的第一条原则是“如果对运算符优先级不确定，就用括号”。当然，要尽量熟悉优先级规则，像 `a*b+c/d` 这种简单表达式还加括号的话，无疑会降低程序的可读性。

可读性为什么这么重要呢？因为程序是给人读的，读者可能是编程者本人，也可能是其他人。丑陋的代码不但会降低程序的可读性和可理解性，而且很难调试正确，因为丑陋的代码往往隐藏逻辑错误。阅读丑陋的代码会更慢，难以让你和其他人相信它是正确的。记

住：绝对不要在程序中使用非常复杂的表达式。例如：

```
a*b+c/d*(e-f/g)/h+7 // 太复杂了
```

另外，变量的命名应该有对应的含义。

4.3.1 常量表达式

程序中经常会用到常量。例如，一个与几何相关的程序会用到 pi，一个英寸到厘米的转换程序会用到转换系数 2.54。显然，常量名应该能够体现它的含义（例如，我们一般用 pi，而不是 3.14159）。而且，常量的值也不应该经常被改变。因此，在 C++ 语言中提供了符号常量来表示那些在初始化后值就不再改变的数值量。例如：

```
constexpr double pi = 3.14159;  
pi = 7; // 错误，试图改变常量的值  
double c = 2*pi*r; // 正确，这里是读 pi 的值，而不是改变它
```

常量对于维护程序的可读性具有重要作用。大部分人可能都知道 3.14159 表示的是 pi，但 299792458 表示什么就没有多少人能猜到了。进一步讲，如果要求程序把 pi 的精度提高到 12 位有效数字，则需要改变程序中所有用到 pi 的语句。一种可行的方法是搜索程序中所有包含 3.14 的地方，但对于使用 22/7 来代替 pi 的语句就搜索不到了。因此，最好的方法是在程序中只有一个定义 pi 的语句，其他用到 pi 的语句都使用该常量。需要修改 pi 值的时候，只修改 pi 的定义语句即可：

```
constexpr double pi = 3.14159265359;
```

因此，我们的建议是：除了个别情况（例如 0 和 1），程序中应该尽量少用字面常量，而是尽可能地使用符号常量。在代码中，这种不能直接被识别的字面常量通常被戏称为魔术常量。

在一些情况下，例如在 case 语句中（4.4.1.3 节），C++ 需要一个常量表达式，即仅由常量构成的整型值表达式。例如：

```
constexpr int max = 17; // 字面常量是常量表达式  
int val = 19;  
  
max+2 // 常量表达式（常量整数加字面常量构成）  
val+2 // 不是常量表达式，因为使用了变量
```

顺便说一下，299792458 是一个基本的物理常量：它是光在真空中的传播速度，单位是 米 / 秒。当你不知道这一点的时候，在代码中看到这个字面常量，肯定会犯糊涂。因此，要避免使用魔术常量。

一个 constexpr 符号常量必须给定一个在编译时已知的值。例如：

```
constexpr int max = 100;  
  
void use(int n)  
{  
    constexpr int c1 = max+7; // 正确，c1 是 107  
    constexpr int c2 = n+7; // 错误，不知道 c2 的值是多少  
    // ...  
}
```

若某个变量初始化时的值在编译时未知，但初始化后也绝不改变，为解决此种情况，C++ 提供了第二种形式的常量 (const)：

```

constexpr int max = 100;

void use(int n)
{
    constexpr int c1 = max+7; // 正确, c1 是 107
    const int c2 = n+7;      // 正确, 但是 c2 的值不能改变
    // ...
    c2 = 7;            // 错误, c2 是常量
}

```

此种“**const** 变量”很常见，原因有两个：

- C++98 不支持 **constexpr**，所以大家以 **const** 替代。
- 不是常量表达式（值在编译时未知）但初始化后不允许改变的“变量”本身就非常有用。

4.3.2 运算符

到目前为止，我们用的都是最简单的运算符，接下来你将会看到许多复杂运算符的使用方法。今后我们将在遇到运算符时加以详细描述，大多数运算符都很好理解。下表给出了常用的运算符：

	名 称	说 明
f(a)	函数调用	a 作为函数 f 的参数
++lval	前缀加	递增 1，并使用递增后的值
--lval	前缀减	递减 1，并使用递减后的值
!a	非	结果是布尔类型
-a	单目减	
a*b	乘	
a/b	除	
a%b	取模	仅用于整型
a+b	加法	
a-b	减法	
out<<b	将 b 写到 out	out 是 ostream 对象
in>>b	从 in 中读取数据存到 b 中	in 是 istream 对象
a<b	小于	结果是布尔类型
a<=b	小于等于	结果是布尔类型
a>b	大于	结果是布尔类型
a>=b	大于等于	结果是布尔类型
a==b	等于	不要与赋值混淆
a!=b	不等于	结果是布尔类型
a&&b	逻辑与	结果是布尔类型
a b	逻辑或	结果是布尔类型
lval=a	赋值	不要与等于混淆
lval*=a	复合赋值	等价于 lval=lval*a, 类似还有 /,%,+,-

上表中的 lval 表示左值，即它可以出现在赋值号左边，在附录 A.5 中有详细介绍。

逻辑运算符 `&&`, `||` 和 `!` 的例子可以分别在 5.5.1 节、7.7 节、7.8.2 节和 10.4 节中找到。

需要注意的是，表达式 `a<b<c` 表示 `(a<b)<c`, `a<b` 的结果是布尔值：`true` 或 `false`。因此，表达式 `a<b<c` 的值等于 `true<c` 或者 `false<c`，而不是 `a<b<c` 表示“`b` 的值是否介于 `a` 和 `c` 之间？”实际上，表达式 `a<b<c` 是没有用处的，在进行比较操作时，千万不要写出这样的表达式。如果在别人的代码中发现了这种表达式，这往往意味着一个错误。

增量表达式至少有三种形式：

```
++a
a+=1
a=a+1
```

哪种方式比较好？为什么呢？建议使用第一种方式 `++a`，它直观地表示了增量的含义，显示了我们要做什么（对 `a` 加 1），而不是怎么做（`a` 加 1，然后结果写到 `a`）。通常，我们认为能够更直接地体现程序思想的编程方式更好一些，因为这种方式更准确，并且更容易被读者理解。假如使用 `a=a+1` 的话，读者可能会想，程序的原意真的是要对 `a` 加 1 吗？不会是要做 `a=b+1`、`a=a+2` 或者 `a=a-1` 但输入出错了吧！而使用 `++a` 方式就不会引起这样的疑问。需要注意的是，上述只讨论程序的正确性和逻辑性，与程序的效率无关。实际上，目前的编译器对 `a=a+1` 和 `++a` 的处理是一样的。同样，我们建议编程时使用 `a*=scale` 而不是 `a=a*scale`。

4.3.3 类型转换

表达式中允许存在不同的数据类型。例如，`2.5/2` 是一个 `double` 类型除以一个 `int` 类型。这表示什么呢？我们应该做整型除法还是双精度浮点型除法呢？整型除法时余数被丢弃，例如 `5/2` 的结果为 2。浮点型除法时余数被保留，例如 `5.0/2.0` 是 2.5。“`2.5/2` 是整型除法还是浮点型除法？”的答案是“浮点型除法，因为整型除法会丢失余数”。也就是说，`2.5/2` 的结果是 1.25 而不是 1。我们遵循这样的规则：如果算术表达式中有 `double` 类型数据的话，就进行浮点型算术计算，结果为 `double` 类型；否则就使用整型算术计算，结果为 `int` 类型。例如：

`5/2` 结果是 2 (不是 2.5) 

`2.5/2` 等同于 `2.5,double(2)`, 结果是 1.25

`'a'+1` 等同于 `int('a')+1`

记号 `type(value)` 和 `type{value}` 表示“将 `value` 转换为 `type` 类型，就像用值 `value` 来初始化 `type` 类型的变量一样”。这意味着，编译器会把上述运算中的 `int` 自动转换为 `double`，或将 `char` 自动转换为 `int`。使用 `type{value}` 可避免窄化转换（3.9.2 节），而 `type(value)` 不能。在运算完成的时候，编译器可能还得再进行一次转换，以用来作为初始化值或赋值语句的右端项。例如：

```
double d = 2.5;
int i = 2;

double d2 = d/i;      // d2 == 1.25
int i2 = d/i;        // i2 == 1
int i3 {d/i};         // 错误: double → int 可能是窄化转换 (3.9.2 节)

d2 = d/i;            // d2 == 1.25
i2 = d/i;            // i2 == 1
```

需要特别注意浮点运算表达式中的整数除法。例如，摄氏温度与华氏温度的转换公式为： $f=9/5 \times c+32$ ，程序代码如下：

```
double dc;
cin >> dc;
double df = 9/5*dc+32; // 注意
```

不幸的是，上述程序不能正确实现摄氏温度与华氏温度的转换功能，因为 $9/5$ 的值是 1 而不是我们期望的 1.8。如果要达到我们期望的结果，必须将 9 或 5（或者两者）转换为 double 类型。

```
double dc;
cin >> dc;
double df = 9.0/5*dc+32; // 正确
```

4.4 语句

4.3 节介绍了利用各种运算符组成表达式来进行相应的数值计算。如果要同时计算多个数值，应该怎么办？如果要重复计算多次呢？如果要在多个可选项中进行选择应如何做？应该如何获得输入、输出数据？和许多语言一样，C++ 语言也是通过语句来实现这些功能的。

到目前为止，我们已经见过两种语句了：表达式语句和声明语句。表达式语句是以分号结束的一个表达式。例如，

```
a = b;
++b;
```

上面是两个表达式语句的例子。注意，`=` 是运算符。因此，`a=b;` 是一个以分号结尾的表达式语句。分号的使用主要是出于技术上的考虑，例如：

```
a = b ++ b; // 语法错误，缺少分号
```

这条语句错误的原因是：如果缺少分号的话，编译器不知道这条语句表示的是 `a=b++;b;` 或者 `a=b;++b;`。这种二义性问题不但存在于编程语言中，也存在于自然语言中。例如，“人吃虎”（man eating tiger）这句话就很令人费解：到底谁吃谁啊？如果加上标点符号就很好理解了：“食人虎”（man-eating tiger）。

计算机是严格按照语句在程序中的书写顺序来执行的，例如：

```
int a = 7;
cout << a << '\n';
```

在这里，变量声明语句及其初始化在输出语句之前执行。

程序中的语句一般都要起作用，我们把不起作用的语句称为无效语句。例如，

```
1+2; // 加法操作，但是程序中没有用到它的结果
a*b; // 乘法操作，但是程序中没有用到它的结果
```

这类无效语句一般都是逻辑错误造成的，编译器会对这类无效语句给出警告信息。总结一下，表达式语句主要包括赋值语句、I/O 语句和函数调用。

此外，我们还介绍一种其他语句形式：空语句。考虑如下代码：

```
if (x == 5);
{ y = 3; }
```

⚠ 上面的语句看上去是有错误的。事实上，按照程序的原意，它也确实存在语义错误：第一行

不应该出现分号结束符。但不幸的是，按照 C++ 语言的语法，这是一个合法的空语句，分号前什么也没有即表示空语句，它什么也不做，很少被使用。但是在上面这个例子中，空语句的存在掩盖了一个语义错误，而且编译器也无法发现这个错误，这样就大大增加了程序员发现错误的难度。

上述代码的执行结果是什么呢？编译器首先会检查 `x` 的值是否等于 5。如果条件是真，那么将执行接下来的语句（空语句），结果是什么也不做。然后程序执行下一行，`y` 被赋值为 3（而按照程序的原意，只有当 `x` 的值等于 5 的时候，才执行赋值操作）。也就是说，这个 `if` 语句没有起作用：无论 `if` 语句的结果是什么，`y` 都被赋值为 3。这是一个新手常犯的错误，而且这种错误很难被发现。

下面一节将讨论可用来更改计算顺序的语句，这样我们就可以表达更有趣的计算，而不仅仅是按照程序编写的顺序逐行执行语句。

4.4.1 选择语句

无论在程序中或者在生活中，我们都会面临各种选择问题。在 C++ 语言中，选择是利用 `if` 语句或者 `switch` 语句实现的。

4.4.1.1 if 语句

`if` 语句是最简单的选择语句，可以在两种可选分支中进行选择。例如，

```
int main()
{
    int a = 0;
    int b = 0;
    cout << "Please enter two integers\n";
    cin >> a >> b;

    if (a<b)      // 条件
        // 第一个分支 (当 if 条件为真时执行)
        cout << "max(" << a << "," << b << ") is " << b << "\n";

    else
        // 第二个分支 (当 if 条件为假时执行)
        cout << "max(" << a << "," << b << ") is " << a << "\n";
}
```

`if` 语句在两个分支之间进行选择，如果条件为真，那么执行第一个分支语句，否则执行第二个分支语句。大部分编程语言都是这样规定的。事实上，编程语言的这种规定来自于实际学习和生活中的习惯。例如，在幼儿园你就应该学习了过马路时要看交通灯“红灯停、绿灯行”，对应的 C++ 程序为：

```
if (traffic_light==green) go();
```

和

```
if (traffic_light==red) wait();
```

虽然 `if` 语句的基本概念很简单，但在使用 `if` 语句时也要仔细。看看下面的程序有什么错误（为了简化，省略了 `#include` 语句）：

```
// 英寸和厘米之间的转换程序
// 后缀 i 和 c 表示输入的单位 (i 表示英寸, c 表示厘米)

int main()
```

```

{
    constexpr double cm_per_inch = 2.54; // 一英寸折合多少厘米
    double length = 1; // 长度，单位是英寸或厘米
    char unit = 0;
    cout << "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;

    if (unit == 'i')
        cout << length << "in == " << cm_per_inch * length << "cm\n";
    else
        cout << length << "cm == " << length / cm_per_inch << "in\n";
}

```

实际上，如果严格按照格式输入数据的话，这个程序是能够正确执行的：输入 1i 得到输出 1in==2.54cm；输入 2.54c 得到输出 2.54cm==1in。读者不妨自己试一下。

这个程序的问题在于我们没有测试非法数据输入的情况，它假定每一次输入都是合法的。程序的条件 `unit == 'i'` 只是区分了 'i' 和其他的所有情况，而没有专门针对 'c' 进行判断。

如果用户输入 15f（15 英尺）会出现什么情况呢？条件表达式 (`unit == 'i'`) 的值为假。因此，程序的 `else` 部分（第二个分支）将被执行，即执行厘米到英寸的转换。但是，这个结果明显不是我们需要的英尺的转换。

 除了合法输入情况以外，程序必须经过各种非法输入的检验。因为对用户来说，非法输入是不可避免的。这些非法输入可能是偶然的，也可能是故意的。但不管用户出于什么目的造成了非法输入的情况，程序都必须能够检测到。

下面给出了上述代码的改进版本：

```

// 英寸与厘米之间的转换程序
// 输入数据后缀 'i' 或 'c' 分别表示输入数据的单位
// 其他后缀是非法的
int main()
{
    constexpr double cm_per_inch = 2.54; // 每英寸折合多少厘米
    double length = 1; // 长度，单位是英寸或厘米
    char unit = ' '; // 单位初值是空格，表示这不是一个单位
    cout << "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;

    if (unit == 'i')
        cout << length << "in == " << cm_per_inch * length << "cm\n";
    else if (unit == 'c')
        cout << length << "cm == " << length / cm_per_inch << "in\n";
    else
        cout << "Sorry, I don't know a unit called " << unit << "\n";
}

```

在这个程序中，依次检验 `unit == 'i'` 和 `unit == 'c'`，如果都不成立，则显示出错信息。看上去好像是我们使用了“`else-if`”语句，但 C++ 语言中没有这种语句。实际上，这里是将两条 `if` 语句组合起来使用的。`if` 语句的一般形式为：

`if (表达式) 语句 else 语句`

关键字 `if` 后面是用括号括起来的表达式，然后是一条语句，`else` 后面是另一个分支语句，其中该分支语句可以是一条 `if` 语句：

`if (表达式) 语句 else if (表达式) 语句 else 语句`

上面所给程序的结构如下：

```

if (unit == 'i')
    ...
else if (unit == 'c')
    ...
else
    ...

```

// 第一个分支
// 第二个分支
// 第三个分支

通过这种方式，我们可以写出包含任意分支的复杂语句。但需要注意的是，代码应该尽量简洁，而不是复杂。编写最复杂的代码并不能显示你的智力水平。反之，能够用简洁的代码完成同样的目标才能体现你的能力。



试一试

基于前面给出的示例程序，编写一个能够将日元、欧元和英镑兑换为美元的程序。如果你注重真实性，可以从互联网上获得最新的汇率。

4.4.1.2 switch 语句

实际上，示例中的 unit 和 'i'、'c' 的比较是最常见的选择形式：基于数值与多个常量比较的选择。在程序设计中经常会用到这种选择，因此 C++ 语言专门提供了一个语句：switch 语句。利用 switch 语句可将前面的程序改写为：

```

int main()
{
    constexpr double cm_per_inch = 2.54; // 每英寸折合多少厘米
    double length = 1; // 长度，单位是英寸或厘米
    char unit = 'a';
    cout << "Please enter a length followed by a unit (c or i):\n";
    cin >> length >> unit;
    switch (unit) {
        case 'i':
            cout << length << "in == " << cm_per_inch * length << "cm\n";
            break;
        case 'c':
            cout << length << "cm == " << length / cm_per_inch << "in\n";
            break;
        default:
            cout << "Sorry, I don't know a unit called '" << unit << "'\n";
            break;
    }
}

```

与 if 语句相比，switch 语句更加清晰易懂，特别是与多个常量进行比较时。关键字 switch 后括号中的值与一组常量进行比较，每个常量用一个 case 语句标记。如果该值与某一常量相等，将选择执行该 case 语句，每个 case 语句都以 break 结束。如果该值与任何一个 case 后的常量都不相等，则选择执行 default 语句。虽然 default 语句不是必需的，但我们建议你加上，除非你能够完全确定给出的分支已经覆盖了所有的情况。注意：编程能够让你理解世界上没有绝对确定的事情。

4.4.1.3 switch 技术

下面是一些与 switch 语句相关的技术细节：

1. switch 语句括号中的值必须是整型、字符型或枚举类型（9.5 节）。特别地，不能使用字符串类型。

2. `case` 语句中的值必须是常量表达式（4.3.1 节），不能使用变量。
3. 不能在两个 `case` 语句中使用相同的数值。
4. 允许在一个 `case` 语句中使用多个 `case` 常量。
5. 不要忘记在每个 `case` 语句末尾加上 `break`。注意：编译器不会给出未加 `break` 的任何警告信息。

例如：

```
int main()           // 你只能将整型等类型用于 switch 语句
{
    cout << "Do you like fish?\n";
    string s;
    cin >> s;
    switch (s) {      // 错误：值必须是整型、字符型或枚举类型
        case "no":
            // ...
            break;
        case "yes":
            // ...
            break;
    }
}
```

如果要对 `string` 类型的数据进行选择，只能使用 `if` 语句或者 `map`（见第 16 章）。

`switch` 语句能够对一组常量的比较产生优化的代码，特别是当常量数目很多的时候，`switch` 语句比 `if` 语句的嵌套使用更加有效。但是，`case` 语句中的值必须是常量，而且不能重复。例如：

```
int main()           // case 标记后的值必须是常量
{
    // 定义分支
    int y = 'y';      // 这将导致问题
    constexpr char n = 'n';
    constexpr char m = '?';
    cout << "Do you like fish?\n";
    char a;
    cin >> a;
    switch (a) {
        case n:
            // ...
            break;
        case y:          // 错误：case 标记的值使用了变量
            // ...
            break;
        case m:
            // ...
            break;
        case 'n':         // 错误：case 标记的值重复使用（变量 n 的值也是 'n'）
            // ...
            break;
        default:
            // ...
            break;
    }
}
```

如果希望采用同样的操作对一组值进行处理，可以为这个操作加上一组标记，而不是重

复写同样的操作代码。例如：

```
int main() // 你可以给一条语句加上多个 case 标记
{
    cout << "Please enter a digit\n";
    char a;
    cin >> a;
    switch (a) {
        case '0': case '2': case '4': case '6': case '8':
            cout << "is even\n";
            break;
        case '1': case '3': case '5': case '7': case '9':
            cout << "is odd\n";
            break;
        default:
            cout << "is not a digit\n";
            break;
    }
}
```



使用 switch 语句时，常犯错误是忘记为 case 语句添加 break。例如：

```
Int main() // 错误代码示例（丢失 break）
```

```
constexpr double cm_per_inch = 2.54; // 一英寸等于 2.54 厘米
double length = 1; // 用英寸或厘米标记的长度
char unit = 'a';
cout << "Please enter a length followed by a unit (c or i):\n";
cin >> length >> unit;

switch (unit) {
    case 'i':
        cout << length << "in == " << cm_per_inch * length << "cm\n";
    case 'c':
        cout << length << "cm == " << length / cm_per_inch << "in\n";
}
```

不幸的是，对于上面这个例子，编译器是不会报错的。当执行完 case 'i' 的代码后，程序接着执行 case 'c' 的代码。如果输入 2i 的话，程序将输出

```
2in == 5.08cm
2cm == 0.787402in
```

这个问题需要特别注意！

试一试

用 switch 语句重写在上一节的“试一试”中给出的汇率转换程序，并且增加人民币和克朗的转换功能。哪一个版本的程序更容易编写、理解和修改呢？为什么？

4.4.2 循环语句

现实生活中，我们经常会遇到一些重复性的工作。为此，编程语言也提供了相应的语言工具，称为循环（repetition）。在对一系列数据进行同样处理的时候，它也被称为迭代（iteration）。

4.4.2.1 While语句

在世界上第一台能存储程序的计算机（名为 EDSAC）上运行的第一个程序就是一个循环语句程序，它是由英国剑桥大学计算机实验室的 David Wheeler 在 1949 年 5 月 6 日编写的，其目的是计算并打印下面这个简单的平方表：

```

0  0
1  1
2  4
3  9
4  16
...
98  9604
99  9801

```

平方表的每一行是一个数，后面跟着一个制表符（'t'），然后是该数的平方。该程序的 C++ 版本如下：

```

// 计算并打印 0 ~ 99 的平方表
int main()
{
    int i = 0;      // 从 0 开始
    while (i < 100) {
        cout << i << '\t' << square(i) << '\n';
        ++i;      // i 值递增（即 i 的值变为 i+1）
    }
}

```

程序中的 `square(i)` 表示 `i` 的平方，其含义和用法将在后续章节中解释（4.5 节）。

不过，这第一个计算机程序并不是用 C++ 编写的，但它们的程序逻辑是相同的，如下所示：

- 从 0 开始计数；
 - 检查计数是否达到 100，如果是的话，程序结束；
 - 否则，打印这个数和它的平方，中间用制表符（'t'）隔开。计数加 1，重复上述操作。
- 显然，完成上述目标，我们需要：
- 一种实现语句重复执行的方法（循环）；
 - 一个记录循环次数的变量（循环变量或控制变量），在上面的例子中是整型变量 `i`；
 - 循环变量的初始化，在示例中是 0；
 - 循环结束条件，在示例中是循环 100 次；
 - 每次循环中完成的操作（循环体）。

这里使用 `while` 语句实现这个功能。在 `while` 语句中，关键字 `while` 之后是循环条件，然后是循环体。

```

while (i < 100)      // 在循环条件中检验控制变量
{
    cout << i << '\t' << square(i) << '\n';
    ++i;              // 控制变量 i 加 1
}

```

循环体是一个程序块，其任务是输出平方表的一行，并将循环控制变量 `i` 的值加 1。每次循环开始都要检查循环条件 `i < 100` 是否成立，若成立则执行循环体；如果不成立，即 `i` 的值达到 100 的时候，则结束 `while` 语句，执行后续的程序代码。在上面的示例中，`while` 语

句是程序的最后一条语句。因此，**while** 语句结束后程序也随之结束。

while 语句的循环控制变量必须在 **while** 语句之前定义和初始化，否则编译器将返回一个错误。如果定义了循环控制变量而没有初始化，大部分编译器会返回一个警告信息“本地变量 *i* 没有赋初值”，但不会作为编译错误来处理。请注意，编译器给出的变量未初始化的信息大都是对的，未初始化的变量会导致很多错误。在上面的示例中，应该加上：

```
int i = 0; // 从 0 开始
```

编写循环语句很简单，但让循环语句能够准确反映实际问题却很困难。其中，主要难点是如何让循环正确地开始和结束，这里的关键是循环条件的设置和所有变量的初始化。

试一试

字符 'b' 可以通过 **char('a'+1)** 得到，字符 'c' 可以通过 **char('a'+2)** 得到。用一个循环语句来实现字母表及其相应 ASCII 码值的输出：

```
a 97
b 98
...
z 122
```

4.4.2.2 程序块

注意下面程序中 **while** 语句的循环体是如何定义的：

```
while (i<100) {
    cout << i << '\t' << square(i) << '\n';
    ++i; // i 值加 1 (即 i 变成 i+1)
}
```

我们把用 { 和 } 包围起来的语句序列称为程序块 (block) 或复合语句 (compound statement)。程序块是一种特殊语句，不包含任何具体语句的程序块也是有用的，它表示什么也不做。例如：

```
if (a<=b) { // 不做任何事
}
else { // 交换 a 和 b
    int t = a;
    a = b;
    b = t;
}
```

4.4.2.3 for 语句

像大多数编程语言一样，C++ 为针对一组数据的迭代操作设置了专门的语句。**for** 语句与 **while** 语句类似，只是 **for** 语句要求将循环控制变量集中放在开头，以便于阅读和理解。使用 **for** 语句的第一个示例如下，

```
// 计算并输出 0 ~ 99 的平方表
int main()
{
    for (int i = 0; i<100; ++i)
        cout << i << '\t' << square(i) << '\n';
}
```

该程序的含义是“从 *i* 等于 0 开始执行循环体，每执行一次循环体，*i* 的值加 1，直到 100 为止”。**for** 语句可以用等价的 **while** 语句来替换，例如：

```
for (int i = 0; i<100; ++i)
    cout << i << '\t' << square(i) << '\n';
```

等价于

```
{
    int i = 0;           // for 语句初始化
    while (i<100) {    // for 语句终止条件
        cout << i << '\t' << square(i) << '\n';
        ++i;             // for 语句增量
    }
}
```

有的初学者喜欢使用 `while` 语句，有的则喜欢使用 `for` 语句。与 `while` 语句相比，`for` 语句的代码更容易被理解和维护。这是因为 `for` 语句将循环相关的初始化、循环条件和增量操作集中放在一起，而 `while` 语句则不是这样。

注意：不要在 `for` 语句的循环体内修改循环控制变量的值。这种操作虽然没有语法错误，但是它违背了读者对于循环的普遍理解和认识。考虑下面这个例子：

```
int main()
{
    for (int i = 0; i<100; ++i) { // for 语句 i 的范围 [0:100)
        cout << i << '\t' << square(i) << '\n';
        ++i; // 这将导致什么结果？似乎是一个错误！
    }
}
```

乍看起来，每个人都认为上面的循环会执行 100 次，但实际上不够 100 次。循环体中的 `++i` 语句使得 `i` 的值每执行一次循环体就加 2，所以只执行了 50 次。这个程序虽然没有语法错误，但是我们看到这样的程序仍然认为程序有错，错误原因也许就是把 `while` 语句转换为 `for` 语句时的粗心大意。如果我们希望循环控制变量每次加 2，可以这么写：

```
// 计算并输出 [0,100) 的偶数的平方表
int main()
{
    for (int i = 0; i<100; i+=2)
        cout << i << '\t' << square(i) << '\n';
}
```

请注意，清晰明了的版本比混乱版本代码要短，通常都是这样。

试一试

使用 `for` 语句重写上一节“试一试”中的字符输出程序，并修改你的程序，使其还可以输出所有大写字母和数字。

也有遍历数据集的更简单的“范围 `for` 循环”，比如 `vector`，见 4.6 节。

4.5 函数

在上面的程序中，`square(i)` 是什么呢？它是一个函数调用。准确地说，它使用参数 `i` 调用 `square` 函数。函数（function）是一个具名的语句序列，能够返回计算结果（称为返回值）。C++ 的标准库提供了许多有用的函数，例如在 3.4 节中用到的求平方根函数 `sqrt()`，但我们

在程序中还需要写很多函数。square 函数的一种可行定义如下：

```
int square(int x) // 返回 x 的平方
{
    return x*x;
}
```

第一行说明这是一个名为 square 的函数（由括号可知），它有一个 int 型参数（名为 x），返回值也是 int 型（函数定义中的第一个关键字）。这个函数的使用如下：

```
int main()
{
    cout << square(2) << '\n'; // 输出 4
    cout << square(10) << '\n'; // 输出 100
}
```

对于函数的返回结果，我们可以使用也可以不使用。但是，我们必须严格按照函数的定义给它传递参数，例如：

```
square(2); // 可能提示信息：“未使用的返回值”
int v1 = square(); // 错误：参数丢失
int v2 = square; // 错误：括号丢失
int v3 = square(1,2); // 错误：参数过多
int v4 = square("two"); // 错误：参数类型错误——应是 int
```

很多编译器都会警告未使用的函数返回值，并给出上面示例中的错误信息。你可能会认为计算机很“聪明”，它应该能够理解“two”表示整数 2。但实际上，C++ 编译器并不像你想象的那样。编译器的工作是检查你的代码是否符合 C++ 语言规范，并严格按照你的程序要求去执行。如果让编译器去猜测你的真实意图的话，那么它很可能猜错，从而导致你或者你的程序用户陷入麻烦。你将会发现如果有编译器的猜测等“帮助”，那么很难预测程序的运行结果。

函数体（function body）是实现某种具体功能的程序块（4.4.2.2 节）。

```
{
    return x*x; // 返回 x 的平方
}
```

函数 square 的实现比较简单：计算参数的平方，并将计算结果作为函数返回值。显然，用 C++ 语言描述比用自然语言（英语、汉语等）描述更简洁。这一点在很多情况下都适用，毕竟，程序语言的目的就是用一种更简洁、准确的方式来描述我们的思想。

函数定义（function definition）的语法描述如下：

类型 函数名（参数表）函数体

其中，类型是函数的返回值类型，函数名是函数的标记，括号内是参数表，函数体是实现函数功能的语句。参数表（parameter list）的每一个元素称为一个参数（parameter）或形式参数（formal argument），参数表可以为空。如果不需要函数返回任何结果，返回值类型可以设置为 void。例如：

```
void write_sorry() // 无参数，无返回值的函数
{
    cout << "Sorry\n";
}
```

函数语法的相关细节可以参考本书第 8 章的内容。

4.5.1 为什么使用函数



当需要将一部分计算任务独立实现的时候，可以将其定义为一个函数，因为这样可以：

- 实现计算逻辑的分离；
- 使代码更清晰（通过使用函数名）；
- 利用函数，使得同样的代码在程序中可以被多次使用；
- 减少程序调试的工作量。

在本书的后续内容中，我们将看到很多解释上述原因的示例，并且还会再次谈及某个原因。注意，实际的应用程序可能会用到成百上千个函数，某些程序甚至会用到上百万个函数。显然，如果这些函数不能被清楚地划分和命名的话，任何人都不可能编写和理解这种包含大量函数的程序。而且，你发现很多函数会被重复使用，很快这将导致你厌倦于这种重复性的劳动。例如，编写处理 x^x 、 7^7 和 $(x+7) \cdot (x+7)$ 等的程序可能会令你高兴。但是，对完成同样功能的函数 `square(x)`、`square(7)` 和 `square(x+7)` 却可能会让你厌倦。这是因为，求平方是非常容易实现的，但对于复杂函数来说，情况大不相同：对于求平方根函数（C++ 中的 `sqrt`）来说，程序员更喜欢使用 `sqrt(x)`、`sqrt(7)` 和 `sqrt(x+7)`，而不是每次都重复实现相同的求平方根的代码（这些代码很长、很复杂）。实际上，大多数情况下，你根本不需要了解求解平方根函数的实现细节，而只需要知道 `sqrt(x)` 将返回 x 的平方根就可以了。

在 8.5 节中，我们将详细介绍相关的函数编写技巧。下面我们将给出另外一个示例。

如果我们想让主函数中的循环更简洁，可将程序改写为：

```
void print_square(int v)
{
    cout << v << '\t' << v*v << '\n';
}

int main()
{
    for (int i = 0; i<100; ++i) print_square(i);
}
```

但我们为什么不用 `print_square()` 版本的程序呢？因为这个版本的程序实际上并不比 `square()` 版本的程序更简洁。因为：

- `print_square()` 是一个比较特殊的函数，以后很难再次用到它，而 `square()` 可以被多次重复使用。
- `square()` 几乎不需要任何额外的说明文档，而 `print_square()` 需要相关文档说明函数的功能和使用方法等。

根本的原因是 `print_square()` 需要执行两个独立的逻辑操作：

- 输出结果；
- 计算平方值。

如果每个函数只完成单一的逻辑操作，那么程序将更易于编写和理解。因此，使用 `square()` 是一个更好的选择。

最后，为什么我们要用 `square()` 而不是第一个版本程序中的 $i*i$ 呢？使用函数的目的之一是用函数把一些复杂的运算分离出来。对程序的 1949 年版本来说，硬件没有提供对乘法操作的直接支持，因此，1949 年版本程序中的乘法是一个类似笔算的复杂计算过程。而且，

1949 年版本程序的作者 (David Wheeler) 也是现代计算中的函数 (称为子程序) 发明人, 这也是使用 `square()` 的原因。

试一试

不用乘法操作实现 `square()` 的功能, 即利用重复加法操作实现 $x \times x$ (设置一个初值为 0 的变量, 把 x 的值加到该变量上 x 次); 然后, 利用这一 `square()` 运行前面的示例。

4.5.2 函数声明

你是否注意到, 函数调用所需的所有信息都已经包括在函数定义的第一行? 例如,

```
int square(int x)
```

根据这些信息, 可以写出如下语句:

```
int x = square(44);
```

我们不需要知道函数体是如何实现的。在编写程序的时候, 我们一般不需要知道函数体的实现细节。为什么我们要知道标准库函数 `sqrt()` 是如何实现的呢? 我们知道它能够计算参数的平方根就够了。为什么我们要阅读 `square()` 函数的代码呢? 虽然我们可能会好奇它的具体实现, 但大多数情况下, 我们仅仅关心如何调用函数就可以了。幸运的是, C++ 提供了一种与函数定义分离的方法来显示函数的信息, 称为函数声明 (function declaration)。

```
int square(int);           // 声明函数 square
double sqrt(double);       // 声明函数 sqrt
```

注意, 函数原型以分号结束, 分号替代了函数定义中的函数体部分:

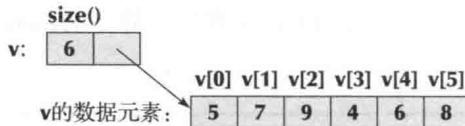
```
int square(int x)          // 定义函数 square
{
    return x*x;
}
```

如果你想使用某个函数, 可以在代码中声明或者通过 `#include` 包含该函数的函数原型, 而函数的定义可以在程序的其他部分, 我们将在 8.3 节和 8.7 节中讨论具体的实现细节。函数原型和函数定义的分离对于大型程序是非常必要的, 我们可以用函数原型来保证代码的简洁, 从而保证在同一时刻将注意力集中在程序的某一局部区域上 (4.2 节)。

4.6 vector

在编写程序之前, 我们首先要准备好相关的数据。例如, 我们可能需要准备一组电话号码, 一个球队的队员表, 一个课程表, 最近一年的读者列表, 下载歌曲的分类表, 汽车付款的可选途径, 下周每一天的天气预测情况, 同一相机在不同网上商店的价格对比表等。程序可能用到的各种数据形式数不胜数, 并存在于程序的所有代码中。我们将从数据的各种存储形式开始介绍 (其他数据存储形式介绍参考本书第 15 和 16 章)。最简单、最常用的数据存储形式是 `vector`。

`vector` 是一组可以通过索引来访问的顺序存储的数据元素。例如, 下图是一个名为 `v` 的 `vector`:



其中，第一个数据元素的索引号是 0，第二个是 1，依此类推。我们可以用 `vector` 名和索引号的组合来表示一个具体的数据元素，例如 `v[0]` 是 5，`v[1]` 是 7，依此类推。`vector` 的索引号总是从 0 开始，每次加 1。看上去有些熟悉，实际上 `vector` 是 C++ 标准库函数中一个历史久远的著名库函数实现的简化版本。图中特别强调了 `vector` “知道自己的大小”，即 `vector` 不仅存储数据元素，也存储元素的个数。

`vector` 可以用如下形式表示：

```
vector<int> v = {5, 7, 9, 4, 6, 8}; // vector 包含 6 个整型数
```

可以看出，定义一个 `vector` 需要确定 `vector` 的数据类型和始集。数据类型在紧跟 `vector` 名的 `<>` 内定义（如 `<int>`）。下面是另一个示例，

```
vector<string> philosopher
    = {"Kant", "Plato", "Hume", "Kierkegaard"}; // 包含 4 个字符串的 vector
```

显然，一个 `vector` 只能存储与其数据类型相同的数据：

```
philosopher[2] = 99; // 错误：试图将一个整型赋给一个字符串
v[2] = "Hume"; // 错误：试图将一个字符串赋给一个整型
```

当一个给定大小的 `vector` 被定义后（但并未指定数据元素值），根据数据类型的不同，它的每一个数据元素将被赋予不同的缺省值。例如：

```
vector<int> vi(6); // vector 的 6 个整型元素初始化为 0
vector<string> vs(4); // vector 的 4 个字符串元素初始化为 ""
```

不含字符的字符串 "" 被称为是空字符串。

注意，不能引用一个不存在的 `vector` 元素。例如：

```
vi[20000] = 44; // 运行时错误
```

我们将在下一章详细讨论关于运行时错误和下标运算的细节。

4.6.1 遍历一个 `vector`

一个 `vector` “知道” 它的大小，所以可以如下打印一个 `vector` 的所有元素：

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int i=0; i<v.size(); ++i)
    cout << v[i] << '\n';
```

函数调用 `v.size()` 返回 `vector` `v` 的元素个数。一般地，`v.size()` 可让我们能访问一个 `vector` 的元素，而不会意外越界。`Vector` `v` 的元素范围是 `[0:v.size()）`，这是数学中的半开序列的记号。`v` 的第一个元素是 `v[0]`，`v` 的最后一个元素是 `v[v.size()-1]`。若 `v.size()==0`，则 `v` 没有元素，是一个空 `vector`。这种半开序列的记号在 C++ 和 C++ 标准库中广泛使用（12.3 节，15.3 节）。

语言本身利用半开序列概念可提供一个简洁的遍历序列元素（比如 `vector` 元素）的方法。例如：

```
vector<int> v = {5, 7, 9, 4, 6, 8};
for (int x : v) // 对每个 vector 的元素 x
    cout << x << '\n';
```

这被称为是“范围 for 循环”，这里“范围”是指“元素序列”。可将 `for (int x : v)` 理解为“对每个 `v` 的整型元素 `x`”，该循环的含义等价于对下标 `[0:v.size())` 进行循环。“范围 for 循环”常用于遍历序列的所有元素且每次只访问一个元素的情形。对于更复杂的循环，如每隔 3 个访问 `vector` 的元素、只访问 `vector` 的后半部分或比较两个 `vector` 的元素等，通常使用更复杂、通用的 for 语句效果会更好（4.4.2.3 节）。

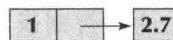
4.6.2 `vector` 空间增长

我们使用 `vector` 的时候，一般是从一个空 `vector` 开始，根据需要逐步填充数据。这里的关键操作是 `push_back()`，它将一个新元素添加到 `vector` 中，该元素成为 `vector` 的最后一个元素。例如：

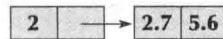
```
vector<double> v; // 初始状态，从空 vector 开始，vectorv 中没有元素
```

v: 

```
v.push_back(2.7); // 在 vectorv 的末尾加入一个值为 2.7 的元素，  
// 现在 v 中包含一个元素，v[0]==2.7
```

v: 

```
v.push_back(5.6); // 在 vectorv 的末尾加入一个值为 5.6 的元素，  
// 现在 v 中包含两个元素，v[1]==5.6
```

v: 

```
v.push_back(7.9); // 在 vectorv 的末尾加入一个值为 7.9 的元素，  
// 现在 v 中包含三个元素，v[2]==7.9
```

v: 

注意 `push_back()` 的调用方法，这是一个成员函数调用（member function call）。`push_back()` 是 `vector` 的一个成员函数，因此它的调用必须采用符号“.”：

成员函数调用：

对象名.成员函数名(参数表)

`vector` 的大小可以通过调用成员函数 `size()` 来获得。初始时 `v.size()` 的值是 0，三次调用 `push_back()` 之后，`v.size()` 的值变为 3。

如果你以前写过程序，你会注意到 `vector` 非常类似于 C 语言或其他编程语言中的数组。但在 `vector` 中，你不需要事先指定 `vector` 的大小，而且可以往 `vector` 中添加任意多个元素。后面还将发现 C++ 标准库的 `vector` 有更多有用的特性。

4.6.3 一个数值计算实例

让我们来看一个更实际的例子。我们经常会遇到把一系列数据读入程序来处理的情况，这些数据处理操作包括：根据数据显示图形，计算平均值和中值，找出最大元素，排序，数据融合，搜索，与其他数据的比较等。对于数据的处理操作是没有任何限制的，但在做各种数据处理前，必须先把数据读入内存中。下面是一种把未知大小（可能很大）的数据读入计算机的基本方法。不失一般性，我们选择读入表示温度的一系列浮点数：

```
// 把温度值读入一个 vector
int main()
{
    vector<double> temps;           // 温度
    for (double temp; cin>>temp; )   // 读操作
        temps.push_back(temp);       // 数据压入 vector
                                    // 其他数据处理
}
```

我们得到什么了呢？首先，我们声明了一个用于存储数据的 `vector`：

```
vector<double> temps; // 温度
```

这条语句说明了我们希望使用的数据类型，此处为 `double` 类型。

接下来是实际的读循环：

```
for (double temp; cin>>temp; )           // 读操作
    temps.push_back(temp);                 // 数据压入 vector
```

这里定义了一个 `double` 类型的变量 `temp`，用来存储读入的温度。语句 `cin>>temp` 读入一个 `double` 类型的数据，然后将这个数据置入 `vector` 中（放在最后）。这里用到的每个操作在前面的示例中几乎都出现过，只是这里使用输入语句 `cin>>temp` 作为 `for` 循环的条件。如果正确输入数据，`cin>>temp` 返回 `true`，否则返回 `false`。因此，`for` 循环将读入我们输入的所有 `double` 类型的数据，直至读到一个其他类型的数据为止。例如，如果输入

1.2 3.4 5.6 7.8 9.0 |

那么 `temps` 将顺序输入 5 个元素 1.2、3.4、5.6、7.8 和 9.0 (`temps[0]==1.2`)。接着，我们用一个字符 `|` 结束输入，实际上，任何一个非 `double` 类型的数据都可以作为输入的结束标志。在 10.6 节中，我们还将讨论如何终止输入和处理输入错误。

为将输入变量 `temp` 的作用域限制在循环内，这里使用了 `for` 语句。也可以使用 `while` 语句：

```
double temp;
while (cin>>temp)           // 读操作
    temps.push_back(temp);   // 数据压入 vector
// ... temp 可能还会用到 ...
```

如前所述，`for` 循环将所需都放在一条语句里，所以代码更易理解，也不容易出错。

一旦数据存入 `vector` 中，就可以方便地对其进行处理。下面的例子用于计算温度的平均值和中值：

```
// 计算温度的均值和中值
int main()
{
    vector<double> temps;           // 温度
    for (double temp; cin>>temp; )   // 读操作
        temps.push_back(temp);       // 数据压入 vector

    // 计算温度的均值
    double sum = 0;
    for (int x : temps) sum += x;
    cout << "Average temperature: " << sum/temps.size() << '\n';

    // 计算温度的中值
    sort(temps);                  // 将 temps 排序
    cout << "Median temperature: " << temps[temps.size()/2] << '\n';
}
```

我们可以用一种简单的方法计算 `vector` 平均值：把所有元素累加到变量 `sum` 中，然后除以元素的个数（即 `temps.size()`）：

```
// 计算温度的均值
double sum = 0;
for (int x : temps) sum += x;
cout << "Average temperature: " << sum/temps.size() << '\n';
```

注意 `+=` 运算符的使用方法。

如果要计算 `vector` 的中值，必须先对 `vector` 进行排序（中值是指序列中大于一半元素而小于另一半元素的那个元素）。我们使用了标准库排序算法 `sort()`：

```
// 计算温度的中值
sort(temps);           // 将 temps 排序
cout << "Median temperature: " << temps[temps.size()/2] << '\n';
```

在第 20 章中我们将解释标准库函数的算法。在排序所有的温度数据后，求中值就很简单了：中值就是下标为 `temps.size()/2` 的那个元素。再深入思考一下，你会发现我们得到的结果有可能不是按照中值定义得到的结果（如果你这么想了，恭喜你，你已经开始像一个程序员一样思考了）。本章的习题 2 将解决这个小问题。

4.6.4 一个文本实例

本节我们不再用温度的例子，因为我们对温度数据并不是特别有兴趣。对于气象学家、农学家、海洋学家等，温度以及基于温度的各类数据是非常重要的。但从程序员的角度看，我们感兴趣的是数据组织的一般形式：可以用于各种应用的 `vector` 以及对 `vector` 的各种操作。总之，不管你对什么内容感兴趣，只要进行数据分析就必须使用 `vector`（或者类似数据结构，具体内容参考第 16 章）。下面的例子说明如何建立一个简单的字典：

```
// 简单字典程序：建立排序单词的列表
int main()
{
    vector<string> words;
    for(string temp; cin>>temp; )      // 读入以空格间隔的单词
        words.push_back(temp);          // 压入 vector
    cout << "Number of words: " << words.size() << '\n';

    sort(words);                     // 对单词排序

    for (int i = 0; i<words.size(); ++i)
        if (i==0 || words[i-1]!=words[i]) // 判断是否是一个新单词
            cout << words[i] << "\n";
}
```

程序会按照字典序输出向程序输入的单词，同时消除重复的单词。例如，输入

a man a plan a canal panama

程序将输出

```
a
canal
man
panama
plan
```

那我们应该如何停止输入呢？或者说，我们应该如何终止这个输入循环呢？

```
for (string temp; cin>>temp; )      // 读
    words.push_back(temp);           // 压入 vector
```

在做数值的读入操作时（参考 4.6.2 节），我们可以通过输入非数值字符来结束输入。但在这个程序中，这种方法是不行的。因为所有的输入字符都被存入一个字符串。幸运的是，我们可以利用一些特殊字符作为终止符。在 3.5.1 节中介绍了 Ctrl+Z 可以终止 Windows 窗口中的一个输入流，Ctrl+D 可以终止一个 Unix 窗口的输入流。

大多数这类程序与前面的温度程序是非常类似的。事实上，我们在写“字典程序”时，很多代码是从“温度程序”中复制过来的，这里只是添加了对单词重复性的判断：

```
if (i==0 || words[i-1]!=words[i])    // 判断是否是一个新单词
```

如果删除这条语句，则输出结果为：

```
a
a
a
canal
man
panama
plan
```

我们不喜欢重复，上面的语句消除了重复性的单词，它是怎样做到的呢？该语句检查前一个已输出的单词是否与下一个即将输出的单词相同 (`words[i-1]!=words[i]`)：如果不同就输出这个单词，否则不输出。显然，在打印第一个单词的时候，不存在前一个单词。因此，还需要判断是否是第一个单词 (`i==0`)。把上面两种判断条件用逻辑或 (`||`) 组合起来可得：

```
if (i==0 || words[i-1]!=words[i])    // 判断是否是一个新单词
```

注意，在比较字符串时可以使用 `!=`（不等于）、`==`（等于）、`<`（小于）、`<=`（小于等于）、`>`（大于）和 `>=`（大于等于）等。这些关系运算符依据字典序进行判断，因此“`Ape`”在“`Apple`”和“`Chimpanzee`”之前。

试一试

编写一个程序，当遇到你不喜欢的单词时会蜂鸣提醒。具体实现思路如下：用 `cin` 输入单词并用 `cout` 输出。如果一个单词属于你预先定义的不喜欢单词的集合，那么程序输出 `BLEEP` 而不是这个单词本身。可以用如下方法定义不喜欢的单词：

```
string disliked = "Broccoli";
```

实现程序功能后，可以尝试增加更多单词。

4.7 语言特性

温度和字典程序用到了本章介绍的大部分语言特性：循环（`for` 语句和 `while` 语句）、选择语句（`if` 语句）、简单的算术运算（`++` 和 `+=` 操作）、比较和逻辑运算（`==`、`!=` 和 `||`）、变量和函数（例如 `main()`、`sort()` 和 `size()` 等）。此外，还用到了标准库提供的功能，例如 `vector`（一种数据元素的容器）、`cout`（标准输出流）和 `sort`（一种排序算法）。

 仔细回想一下，你会发现我们已经用到了很多语言特性。每一种语言特性都表示了一种基本思想，将许多种语言特性结合起来，我们就能写出有用的程序了。记住：计算机不是一

种只能完成固定功能的设备，它可以完成我们能想象到的任何计算任务。而且，通过计算机与其他设备的结合，理论上我们可以用它来完成任何任务。

简单练习

请逐步完成下列练习，不要贪快而跳过这些简单练习。请至少使用三组不同的数据来测试这些程序，鼓励使用更多的测试数据。

1. 编写一个使用 `while` 循环语句的程序，每次循环输入两个 `int` 数据并输出它们，当输入 `#` 后程序结束。
2. 修改程序，输出 “`the smaller value is :`” 后接着输出较小的整数，输出 “`the larger value is:`” 后接着输出较大的整数。
3. 改进程序：当两个整数相等时，输出 “`the numbers are equal`”。
4. 改变程序：输入的数据改为 `double` 型而不是 `int` 型。
5. 改变程序：依次输出较大和较小的数。如果两个数的差小于 `1.0/100` 的话，再输出 “`the numbers are almost equal`”。
6. 改变循环体程序：每次循环只输入一个 `double` 型数据，并用两个变量记录到目前为止的最小数和最大数。每次循环输出当前输入的数据，如果这个数据是到目前为止最小或最大的数，在其后分别输出 “`the smallest so far`” 和 “`the largest so far`”。
7. 为每个 `double` 型数据增加单位，例如数据可以是 `10cm`、`2.5in`、`5ft` 或 `3.33m`。程序可以接受四种计量单位 `cm`、`m`、`in` 和 `ft`，假定转换系数是 `1m==100cm`, `1in==2.54cm`, `1ft==12in`。将单位读入一个字符串。考虑 `12 m` (数字和单位之间有空格) 等价于 `12m` (没有空格)。
8. 改进程序使之拒绝没有单位或单位非法的数据，例如 `y`、`yard`、`meter`、`km` 和 `gallons` 等。
9. 除了记录到目前为止最大和最小的数据外，还记录到目前为止的数据累加和以及数据个数。循环结束后，输出最小、最大、数据个数以及累加和。注意，因为计算累加和必须使用同一计量单位，所以需要事先决定使用哪个计量单位，例如米。
10. 将上述所有的数据（单位转换为米）存入一个 `vector` 变量，然后输出这些数据。
11. 在输出 `vector` 中的数据前，按照升序将这些数据排序。

思考题

1. 什么是计算？
2. 什么是计算的输入和输出？举例说明。
3. 当表示计算的时候，程序员需要谨记哪三项要求？
4. 什么是表达式？
5. 在本章内容中，表达式和语句有什么区别？
6. 什么是左值？列出要求用左值的运算符。为什么这些运算符需要使用左值？
7. 什么是常量表达式？
8. 什么是字面常量？
9. 什么是符号常量，我们应该如何使用它？
10. 什么是魔术常量？举例说明。
11. 哪些运算符既可以用于整型也可以用于浮点型？
12. 哪些运算符只能用于整型而不能用于浮点型？

13. 哪些运算符可以用于字符串？
14. 什么情况下程序员更喜欢用 `switch` 语句而不是 `if` 语句？
15. `switch` 语句常见的问题有哪些？
16. `for` 循环语句的循环控制中每一部分的功能是什么？它们的执行顺序是怎样的？
17. 什么情况下应该使用 `for` 循环？什么情况下应该使用 `while` 循环？
18. 如何输出一个字符型数据的数值？
19. 函数原型 `char foo(int x)` 的含义是什么？
20. 什么情况下你将在程序中定义一个单独的函数？列出原因。
21. 哪些操作可以用于整型数据而不能用于字符串？
22. 哪些操作可以用于字符串而不能用于整型数据？
23. `vector` 中第三个元素的索引号是多少？
24. 如何用 `for` 循环来打印输出 `vector` 的所有数据元素？
25. 语句 `vector <char> alphabet(26);` 的含义是什么？
26. 描述 `vector` 中 `push_back()` 的含义。
27. `vector` 的成员函数 `size()` 的功能是什么？
28. 为什么 `vector` 被如此广泛地使用？
29. 如何将一个 `vector` 中的数据排序？

术语

abstraction (抽象)	loop (循环)
<code>begin()</code>	<code>lvalue</code> (左值)
computation (计算)	<code>member function</code> (成员函数)
conditional statement (条件语句)	<code>output</code> (输出)
declaration (声明)	<code>push_back()</code>
definition (定义)	<code>range-for-statement</code> (范围 <code>for</code> 语句)
<code>divide and conquer</code> (分治)	<code>repetition</code> (重复)
<code>else</code>	<code>rvalue</code> (右值)
<code>end()</code>	<code>selection</code> (选择)
expression (表达式)	<code>size()</code>
<code>for</code> -statement (<code>for</code> 语句)	<code>sort()</code>
function (函数)	<code>statement</code> (语句)
<code>if</code> -statement (<code>if</code> 语句)	<code>switch</code> -statement (<code>switch</code> 语句)
increment (增量)	<code>vector</code>
input (输入)	<code>while</code> -statement (<code>while</code> 语句)
iteration (迭代)	

习题

1. 如果你还没有完成本章中的“试一试”，请先完成相关练习。
2. 定义一个序列的中值恰好是序列的一半元素在它之前而另一半元素在它之后的数值，修改 4.6.3 节的程序使其总是能够输出中值。提示：中值并不一定是序列中的元素。

3. 输入一组 `double` 型数据到一个 `vector` 中，假设这些数据是沿着某一条路径的相邻城市间的距离。要求：计算并输出全部距离（所有距离的总和）；搜索并输出相邻两个城市间的最小和最大距离；搜索并输出两个相邻城市间的平均距离。
4. 编写一个猜数游戏程序。用户给出一个 1 到 100 之间的整数，程序通过提问来猜测用户所想的数是什么（例如，“你的数小于 50 吗？”），程序应该能够用不超过 7 个问题来确定这个数。提示：使用 `<` 和 `<=` 运算符以及 `if-else` 语句编写程序。
5. 实现一个简单的计算器程序。计算器应该能够对两个输入数据实现五种基本数学操作：加、减、乘、除和取模（余数）。程序应该提示用户输入三个参数：两个 `double` 型数据和一个表示操作的字符。例如，如果输入参数 35.6, 24.1 和 '+'，程序将输出“`The sum of 35.6 and 24.1 is 59.7`”。在第 6 章我们将介绍一个更复杂的计算器程序。
6. 定义一个能够存储 10 个字符串的 `vector`，分别是“`zero`”，“`one`”，…“`nine`”。编写一个能够实现数字与其对应拼写进行转换的程序。例如，当输入 7 的时候输出 `seven`。同时，该程序还能够实现拼写形式到数字形式的转换，例如，当输入 `seven` 的时候输出 7。
7. 修改习题 5 的“迷你计算器”程序，使程序不但能够接受数字形式的数据，也能够接受拼写形式的数据。
8. 有一个古老的故事讲述的是一个皇帝为了感谢国际象棋的发明人，答应这个发明人可以提出自己的赏赐要求。发明人提出的要求是：在棋盘的第一个格子里放 1 粒米，在第二个格子里放 2 粒米，第三个格子里放 4 粒米，依次类推。每次都加倍直到放满棋盘的所有 64 个格子为止。这个要求听起来很谦虚，但实际上全国所有的米都不够支付这个赏赐。编写程序来计算一下发明人要获得至少 1000 粒米需要多少个棋盘格子？至少 1 000 000 粒米呢？至少 1 000 000 000 粒米呢？当然，你需要设计一个循环，也许还需要一个整型变量记录当前所处的格子，一个整型变量记录当前格子的米数，一个整型变量记录以前所有格子的米数。建议你在每次循环中都输出所有的变量值，并观察一下会发生什么情况。
9. 尝试计算习题 8 中发明人要求的大米的总量是多少？你会发现这个数字是如此大，以至于 `int` 或 `double` 都无法保存。注意观察当数值太大导致 `int` 或 `double` 无法保存时会发生什么。如果使用整型的话，你可以准确计算出大米总量的最大棋盘的格子数目是多少？使用 `double` 呢？
10. 编写一个“石头、剪刀、布”的游戏程序。如果你不熟悉这个游戏，可以先调查一下（例如使用 Google）。对于程序员来说，调查是日常工作的一部分。用 `switch` 语句解决这个习题。程序将随机给出下一个操作（即石头、剪刀或布是随机出现的）。目前，真正的随机性是很难实现的，因此在程序中可以用存有一个数据序列的 `vector` 来处理，其中 `vector` 的每个数据元素表示程序的下一个操作。根据这个 `vector` 中的数据元素，程序的每次运行将执行相同动作，因此你需要用户输入某些值。尝试做一些变化，使得用户很难猜出程序的下一个动作是什么。
11. 编写程序找出 1 到 100 之间的所有素数。一种可行的方法是用一个函数来判断一个数是否是素数（即判断一个数是否能够被小于它的素数整除），这里需要用到一个 `vector` 类型的素数表（例如，如果这个 `vector` 变量名为 `primes`, `primes[0]==2`, `primes[1]==3`, `primes[2]==5` 等）。然后用一个 1 到 100 的循环逐个判断每个数是否是素数，并将其中的素数存储在一个 `vector` 中。然后，编写另一个循环来显示所有的素数。你可以比较一下你找到的素数和素数表 `primes` 的结果。其中，2 是第一个素数。

12. 修改上面的习题，要求程序有一个输入值 `max`，并找出从 1 到 `max` 的所有素数。
13. 使用名为“埃拉托色尼筛法”（Sieve of Eratosthenes）的经典方法，编写程序找出 1 到 100 之间的所有素数。如果不了解这个方法，你可以通过互联网搜索相关资料。
14. 修改上面的习题，要求程序有一个输入值 `max`，并找出从 1 到 `max` 的所有素数。
15. 编写程序，要求：有一个输入值 `n`，输出结果是前 `n` 个素数。
16. 编写程序，要求：能够找出一组输入数据中最大和最小的数据。在一组数据中出现次数最多的数称为 `mode`。要求：输入一组正整数，程序能够找出该组数据的 `mode`。
17. 编写程序，要求：输入一组字符串，找出该组字符串中最大、最小和 `mode` 字符串。
18. 编写解一元二次方程的程序。一元二次方程的一般形式为

$$ax^2+bx+c=0$$

如果你不知道如何解一元二次方程，可以先调查一下。记住，在一个程序员教会计算机如何解决问题前，程序员必须先清楚如何解决它。程序的输入数据 `a`、`b` 和 `c` 为 `double` 型；一元二次方程有两个解，因此程序的输出包括两个解 `x1` 和 `x2`。

19. 编写程序，其输入是一组名字和数值对。例如，`Joe 17` 和 `Barbara 22` 等。对于每一个名字 - 数值对，名字存入名为 `names` 的 `vector` 中，数值存入名为 `scores` 的 `vector` 对应位置中（例如，如果 `names[7] = "Joe"`，那么 `scores[7] = 17`）。当输入 `No more` 时，终止输入（这里 `more` 将导致读入一个整型数据的失败）。注意，要检查名字的唯一性，相同的名字将导致程序中断并输出一个错误信息。最后，按照每行一个（名字，数值）对的形式输出所有数据。
20. 修改习题 19 的程序，当你输入一个名字后，程序将输出相应的成绩或者输出“`name not found`”。
21. 修改习题 19 的程序，当你输入一个整数后，程序将输出所有对应的名字或者“`score not found`”。

附言

从哲学的观点看，用计算机能做的所有事情，你现在都可以做了，剩下的就是一些具体细节了。由于你是一个编程的初学者，我们要郑重地提醒你：各种编程的细节和技巧对你来说非常重要。通过本章的内容，你已经练习了许多计算技巧：各种变量的使用（包括 `vector` 和 `string`），算术和比较运算符的使用，选择和循环语句等。利用这些基本单元，你可以完成许多计算任务。你也练习了文本和数字的输入和输出，所有的输入和输出都可以表示为文本形式（包括图形）。利用一系列函数，你可以很好地组织你的程序。接下来你需要完成的任务是写好代码，即你的程序要正确、可维护和高效。更重要的是，你要踏踏实实地努力学习如何写好程序。

错 误

我已经意识到从现在开始我的大部分时间将花在寻找和纠正自己的错误中。

——Maurice Wilkes, 1949

在本章内容中，我们将讨论程序的正确性、错误和错误处理。如果你是一个新手，你会发现这种讨论有时有点抽象，有时又显得过于细节化了。错误处理真的很重要么？是的，它非常重要。在你写出别人愿意使用的程序前，你需要掌握几种错误处理方法。我们要做的是向你展示如何“像一个程序员一样思考”。它是建立在对细节和替代方案细致分析基础上的各种抽象策略的组合。

5.1 简介

在前面章节中的练习和习题中，我们已经多次提到了错误处理的相关内容，对于错误你应该已经有了一些初步的认识。在编写程序的时候，错误是不可避免的。当然，最后的程序必须是没有错误的，至少不存在我们不可接受的错误。

错误的分类有很多种，例如，

- 编译时错误：由编译器发现的错误。根据所违背的语法规则，编译时错误还可以进一步细分，例如：
 - 语法错误；
 - 类型错误。
- 链接时错误：当链接器试图将对象文件链接为可执行文件时发现的错误。
- 运行时错误：程序运行时发现的错误。运行时错误可以被进一步细分为，
 - 由计算机（硬件或操作系统）检测出的错误；
 - 由库（例如标准库）检测出的错误；
 - 由用户代码检测出的错误。
- 逻辑错误：由程序员发现的会导致不正确结果的错误。

理想情况下，程序员的任务是消除所有的错误。但在实际中，这经常是不可行的。事实上，对于一个实际程序来说，如何准确定义“所有错误”都是很困难的。如果我们把一台正在执行程序的计算机的电源线拔掉，那么这会是一种你认为的错误么？在多数情况下，答案显然是否定的。但是，如果我们讨论的是医疗设备的监控程序或者电话交换机的控制程序的话，用户会认为包括程序在内的整个系统出了问题。用户只关心结果，而不关心导致这种情况的原因是计算机断电，还是宇宙射线损坏了存放程序的存储器。因此，问题转化为“我们的程序能够检测到错误么？”除非特别说明，我们会假定你的程序：

1. 对于所有合法输入应输出正确结果。
2. 对于所有非法输入应输出错误信息。
3. 不需要关心硬件故障。

4. 不需要关心系统软件故障。
5. 发现一个错误后，允许程序终止。

假设 3、4 和 5 不成立的程序超出了本书的内容范围。但是，假设 1 和 2 是属于程序员的基本专业能力范畴。而培养这种专业能力正是我们的目标之一。即使在实际中，我们不能 100% 达到理想目标，但它是我们的努力方向。

在我们编写程序的时候，出现错误是很自然的和不可避免的。问题是应该如何处理错误？我们估计在开发正式软件时，90% 以上的工作是放在如何避免、查找和纠正错误上。对于一些高可靠软件来说，这一比例甚至要更高。对于小程序来说，你可以把错误处理做得很好。但是，如果你很马虎，你也可能做得很糟糕。

总之，我们有三种方法来编写可接受的软件：

- 精心组织软件结构以减少错误；
- 通过调试和测试，消除大部分程序错误；
- 确定余下的错误是不重要的。

上述任何一种方法都不能保证完全消除错误。我们必须同时使用上述三种方法。

在开发可靠软件时，经验往往会起巨大作用。这意味着，根据用途的不同，程序可以运行在可以接受错误率下。请不要忘记，理想情况下，程序总是做正确的事。虽然我们一般只能接近这一理想情况，但这不是我们不努力工作的借口。

5.2 错误的来源

错误的来源包括：

- 缺少规划：如果没有事先规划好程序要做什么，我们不可能充分检查所有“死角”，并确认所有的可能情况都会被正确处理（即对任意输入程序都能给出正确结果或者充分的错误信息）。
- 不完备的程序：在软件开发过程中，显然会有一些情况我们没有考虑到，这是不可避免的。我们必须达到的目标是掌握我们何时能够处理所有情况。
- 意外的参数：函数会使用参数。如果为一个函数输入了一个不能处理的参数的话，我们会遇到问题。例如，为求平方根的标准库函数输入 -1.2 : `sqrt(-1.2)`。由于 `sqrt()` 的输入和输出都是 `double`，在这种情况下，它不可能输出正确结果。5.5.3 节将讨论这种情况。
- 意外的输入：典型的程序输入包括键盘、文件、图形界面和网络等。对于这些输入，程序一般都设定了许多前提假设。例如，用户会输入一个数字。但是，如果用户输入的是“喂，闭嘴！”而不是期望的数字，程序会怎样呢？5.6.3 节和 10.6 节将讨论这类问题。
- 意外状态：多数程序都保留有很多系统各个部分所使用的数据（“状态”）。例如，地址表、电话簿、温度记录等。如果这些数据是不完整的或者错误的，那应该如何处理呢？无疑程序的各个部分仍然应该正常运转。26.3.5 节将讨论这类问题。
- 逻辑错误：这表示程序没有按照我们所期望的那样运行。我们不得不查找并修正这些问题。在 6.6 节和 6.9 节中我们将给出这类问题的例子。

上述这一列表可以用于实际开发。当我们在开发一个软件的时候，可以把上述列表作为检查表。在我们认为已经排除了所有潜在错误来源之前，软件是不能被交付使用的。实际上，从项目开始时，我们就始终要关注错误处理。因为，没有认真考虑过错误处理的软件几

乎是不可能正常工作的，它还会被重新编写一遍。

5.3 编译时错误

在写程序的时候，编译器是检查错误的第一道防线。在生成可执行文件之前，编译器通过分析代码来检查语法和类型错误。只有编译器认为代码完全符合语法规范，编译过程才能继续下去。编译器发现的大部分错误都很简单，属于“低级错误”。它们一般是由源码的编辑错误导致的。其他一些问题则是由程序各个部分之间的交互引起的。作为初学者，你可能会觉得编译器很繁琐。但是当你学会使用一些语言特性（特别是类型系统）来直接表达你的思想时，你将会认识到编译器的错误检查功能的价值。如果没有编译器，乏味的除错工作将会花费你大量时间。

举例来说，下面是一个简单的函数调用：

```
int area(int length, int width); // 计算一个矩形的面积
```

5.3.1 语法错误

如果我们按照如下方式调用 `area()`，会有什么结果呢：

```
int s1 = area(7; // 错误：)丢失
int s2 = area(7) // 错误：;丢失
Int s3 = area(7; // 错误：Int 不是数据类型
int s4 = area('7); // 错误：非终止符 ('丢失)
```

上面每一行程序都有一个语法错误，即它们不符合 C++ 语言的语法规范，因此编译器会拒绝它们。不幸的是，对你，即程序员来说，理解语法错误的报告信息往往不是那么容易。为了确定错误，编译器往往会读取更多的信息。这会导致即使是一个小错误（往往在发现这个错误时，你会觉得不可思议，自己怎么会犯这种低级错误），编译器也会报告很多繁杂信息，甚至会指向程序中的其他行。因此，如果你在编译器所指向的错误行中没有发现错误的话，还应该检查一下前几行程序是否有错。

需要注意的是，编译器并不知道你想做什么。它只会报告你所做的是否有错，而不会报告你想做的是是否有错。例如，在上面的例子中，`s3` 的声明有错，但是编译器不会告诉你：

“你拼错了 `int`, `i` 不要大写”

而是报告如下信息：

“语法错误：变量 `s3` 前丢失 `;`”

“`s3` 没有存储类型或数据类型”

“`Int` 没有存储类型或数据类型”

在你习惯并理解这些信息含义前，这些信息是很令人费解的。对于同一代码，不同的编译器可能会给出不同的错误信息。幸运的是，你会很快习惯理解这些信息的。实际上，上面这些令人费解的信息可以被解释为：

“在 `s3` 前有一个语法错误，需要检查一下 `Int` 或者 `s3` 的数据类型”

实际上，发现这些问题并不是一件很困难的事。

试一试

尝试编译一下上面的例子，看看编译器的返回信息是什么。

5.3.2 类型错误

一旦语法错误被排除后，编译器就会开始检查类型错误：它将会检查你所声明的变量和函数的类型（或者发现你忘记了声明类型）；检查赋予变量或函数的数值或表达式的类型以及传递给函数参数的数值或表达式的类型。例如：

```
int x0 = arena(7);           // 错误：未声明函数
int x1 = area(7);            // 错误：参数个数不匹配
int x2 = area("seven",2);    // 错误：第一个参数的类型不匹配
```

让我们仔细分析一下这些错误：

- 对于 `arena(7)`，我们将 `area` 错写为 `arena`。因此编译器认为我们是要调用函数 `arena`。（编译器还有其他“想法”吗？这就是前文所说的，编译器是不知道你想做什么的。）假设没有函数 `arena()`，你会得到未定义函数的错误信息。如果存在函数 `arena()` 并且这个函数能够接受 7 作为输入参数的话，你会遇到一个更大的麻烦：程序将会上被正确编译，但是它不会按照你预想的那样去运行（这是一个逻辑错误，详见 5.7 节）。
- 对于 `area(7)`，编译器检查到错误是参数个数不匹配。对于 C++ 语言，函数调用必须使用正确的参数个数、参数类型和顺序。在合理使用类型检查系统时候，它可以作为实时的错误检查工具（详见 19.1 节）。
- 对于 `area("seven",2)`，你可能期望编译器能够识别出 "seven" 表示的是数字 7。但它做不到这点。当一个函数需要输入一个整数的时候，我们不能给它一个字符串。C++ 确实支持一些隐含的类型转换（见 3.9 节），但不包括 `string` 到 `int` 的转换。编译器不会试图去猜测你所要表示的含义。不然，你认为 `area("Hovel lane",2)`, `area("7,2")` 和 `area("sieben","zwei")` 表示什么含义呢？

上述只是一些简单的示例。编译器能够发现更多的错误信息。



试一试

尝试编译一下上面的例子，看看编译器的返回信息是什么。尝试考虑一下你所遇到的更多的错误信息。

5.3.3 警告

当你使用编译器的时候，你会希望它足够聪明，能够理解你要表达的意思，也就是说，你可能会希望编译器报告的一些错误并不是真正的错误。这种想法是很自然的。令人惊奇的是，当你有了一定编程经验后，你会希望编译器能够拒绝更多代码，而不是更少。看下面的例子：

```
int x4 = area(10,-7);      // 正确：但是矩形的宽怎么会是 -7?
int x5 = area(10.7,9.3);   // 正确，但实际上调用的是 area(10,9)
char x6 = area(100,9999);  // 正确，但结果越界了
```

对于 `x4`，我们没有从编译器得到错误信息。对于编译器来说，`area(10,-7)` 是正确的：`area()` 需要两个整型参数，你给定了两个，而没有人规定参数必须是正数。

对于 `x5` 来说，好的编译器将给出警告信息：`double` 型数 10.7 和 9.3 被截取为 `int` 型数

10 和 9 (见 3.9.2 节)。然而, (老的) 语法规则认为可以隐式地将 double 转换为 int, 因此编译器不会拒绝函数调用 area(10.7, 9.3)。

对 x6 的初始化存在与 area(10.7, 9.3) 同样的问题。area(100,9999) 的 int 型返回值, 即 999900, 被赋给一个 char 型变量。x6 最有可能的结果是“截取值” -36。同样, 一个好的编译器将会报告相关的警告信息, 即便 (老的) 语法规则不拒绝相关代码。

当获得一定编程经验后, 你将学会如何脱离编译器去检查错误和避免编译器的弱点。但是, 不要过分自信: “程序已编译” 并不意味着它能够运行。即使它能够运行, 在你消除逻辑错误前, 通常程序给出的也是错误的结果。

5.4 链接时错误

一个程序一般包括几个独立的编译部分, 称为编译单元。程序中每一个函数在所有编译单元中的声明类型必须严格一致。我们使用头文件来保证这一点, 详见 8.3 节。并且, 程序中的每个函数只能定义一次。如果上述两条规则的任意一条被违反的话, 链接器将报错。如何避免链接错误将在 8.3 节中讨论。下面是一个典型的程序链接错误示例:

```
int area(int length, int width); // 计算矩形面积

int main()
{
    int x = area(2,3);
}
```

除非我们在另一个源文件中定义了 area(), 并且将这一源文件的编译单元与当前文件链接, 否则的话, 链接器将报告没有发现 area() 的定义。

同时, area() 的定义必须与我们的调用具有严格相同的类型 (包括返回值类型和参数类型) 即:

```
int area(int x, int y) /* ... */ // “我们的” area()
```

具有相同名称但是类型不同的函数将不会被匹配上, 并被忽略:

```
double area(double x, double y) /* ... */ // 不是“我们的” area()
```

```
int area(int x, int y, char unit) /* ... */ // 不是“我们的” area()
```

需要注意的是, 函数名的拼写错误并不总会导致链接错误。实际上, 当遇到一个未声明函数被调用时, 编译器将立刻报告错误信息。这一点很好: 编译错误早于链接时错误被发现有助于错误的及早排除。

正如上文所述, 函数的链接规则同样适用于程序的其他实体, 例如变量和类型: 具有同一名字的实体只能有一个定义, 但是可以有多个声明, 所有声明必须具有相同的类型。详见 8.2 ~ 8.3 节。

5.5 运行时错误

如果你的程序没有编译错误和链接错误的话, 那么它就能运行了。现在才是乐趣的真正开始。在你写程序的时候, 你能够轻易地发现错误。但是, 对于运行程序时发现的错误, 你可能很难确定如何解决它, 例如:

```
int area(int length, int width) // 计算矩形面积
{
```

```

    return length*width;
}

int framed_area(int x, int y)      // 计算内框面积
{
    return area(x-2,y-2);
}

int main()
{
    int x = -1;
    int y = 2;
    int z = 4;
    ...
    int area1 = area(x,y);
    int area2 = framed_area(1,z);
    int area3 = framed_area(y,z);
    double ratio = double(area1)/area3;    // 强制转换为 double 类型
}

```

在程序中，我们使用变量 `x`, `y`, `z` (而不是直接使用数值作为参数)，这使得阅读代码的人和编译器更难发现问题。但是，这些调用会把负数赋给 `area1` 和 `area2`，导致面积表示为负数。我们能够接受这种明显违背数学和物理规律的错误结果么？如果不能，应该由谁来检测这种错误：`area()` 的调用者或者函数本身？这种错误又应该如何被报告呢？

在回答这些问题之前，我们先看看上面代码中 `ratio` 的计算。这条语句看上去没有什么问题。但你是否发现了什么不对的地方？如果没有，再仔细看看：`area3` 的值是 0，因此 `double(area1)/area3` 将除以 0。这会导致一个硬件检测错误并终止程序，同时报出一个与硬件相关的错误信息。这类错误就是运行时错误。如果你或者你的用户没有及时发现并解决这类错误，在程序运行时这类错误就可能出现。对于这类“硬件错误”，大多数人的容忍度都很低。因为他们不了解程序的细节而仅仅知道“某个地方出了问题”。这点信息对于处理问题帮助很小。在这种情况下，人们会很生气并对程序的提供者抱怨连连。

让我们再检查一下 `area()` 的参数错误问题。我们发现有两个解决办法：

- 让 `area()` 的调用者来处理不正确的参数。
- 让 `area()` (函数本身) 来处理不正确的参数。

5.5.1 调用者处理错误

先看看第一种方法（“让用户意识到问题”）。如果 `area()` 是一个由我们不能修改的库提供的函数，那么我们将选择这种方法。不论好坏，这都是一个合适的选择。

在 `main()` 函数中保护 `area(x,y)` 的调用是很容易的：

```

if (x<=0) error("non-positive x");
if (y<=0) error("non-positive y");
int area1 = area(x,y);

```

事实上，当你发现一个错误后，唯一的问题就是如何解决它。这里我们调用了一个函数 `error()`。它能够做一些错误处理工作。实际上，在 `std_lib_facilities.h` 中我们提供了一个 `error()` 函数。它能够终止程序运行并将字符串参数作为系统错误信息输出。如果你希望输出自己的错误信息并做其他的操作，参看 `runtime_error` (5.6.2 节, 7.3 节, 7.8 节和附录 B.2.1)。对于初学者来说，这已经足够了。它还可以作为更复杂错误处理的实例。

如果我们不需要明确区分每一个参数，我们还可以简化程序如下：

```
if (x<=0 || y<=0) error("non-positive area() argument"); // || 表示逻辑或
int area1 = area(x,y);
```

为了对 area() 的参数实现完全保护，我们需要使用 framed_area() 函数，程序改写为：

```
if (z<=2)
    error("non-positive 2nd area() argument called by framed_area()");
int area2 = framed_area(1,z);
if (y<=2 || z<=2)
    error("non-positive area() argument called by framed_area()");
int area3 = framed_area(y,z);
```

这看上去有些混乱，而且还存在一些基本问题。上面的程序只有在我们确切了解 framed_area() 如何使用 area() 的情况下才是正确的。我们要知道 framed_area() 对每一个参数都减了 2。我们不得不了解这么多细节情况。如果有人把 framed_area() 修改为减 1 而不是 2，我们又该怎么办呢？如果这种情况发生的话，我们不得不查找程序中的每一个 framed_area() 调用，并做相应的修改。这被称为“易碎”代码，应为它很容易被破坏。这也是一个“魔术常量”的例子（4.3.1 节）。为了减少程序的“易碎性”，我们可以在 framed_area() 中用一个命名常量代替具体的数值：

```
constexpr int frame_width = 2;
int framed_area(int x, int y) // 计算内框面积
{
    return area(x-frame_width,y-frame_width);
}
```

这个常量也可以被 frame_area() 的调用者使用：

```
if (1-frame_width<=0 || z-frame_width<=0)
    error("non-positive argument for area() called by framed_area()");
int area2 = framed_area(1,z);
if (y-frame_width<=0 || z-frame_width<=0)
    error("non-positive argument for area() called by framed_area()");
int area3 = framed_area(y,z);
```

仔细看看上面的代码，你能确定它是正确的么？它形式上漂亮么？它易读么？事实上，我们发现它很糟糕（因此容易出错）。我们将代码长度增加了三倍，并且还不得不去了解 frame_area() 的实现细节。但是我们仍然没有达到目的。应该有更好的办法解决这一问题！

再看看原始代码：

```
int area2 = framed_area(1,z);
int area3 = framed_area(y,z);
```

这段代码也许会出错，但我们至少知道它要做什么。我们可以留用这段代码，而把错误检查移到 framed_area() 内部。

5.5.2 被调用者处理错误

在 framed_area() 内部实现错误检查非常简单，error() 仍然可以被用于错误报告：

```
int framed_area(int x, int y) // 计算内框的面积
{
    constexpr int frame_width = 2;
    if (x-frame_width<=0 || y-frame_width<=0)
        error("non-positive area() argument called by framed_area());
```

```

    return area(x-frame_width,y-frame_width);
}

```

这一实现非常好，而且我们也不用为每一个 `frame_area()` 调用写一个测试。在一个大程序中，对一个会被调用 500 次的常用函数来说，这一点非常有用。而且，如果需要对错误处理进行修改的话，我们只需要在一处地方进行改动就可以了。

需要注意的是，在这里我们很自然地从“调用者必须检查参数”的方法转变到“函数必须检查自己的参数”的方法（也称为“被调用者检查”）。后者的好处在于参数检查只在一个地方实现。我们不需要在整个程序中查找调用点。而且，参数检查只在这一一个地方实现，我们可以方便地掌握参数检查的全部信息。

让我们把这一思想应用于 `area()` 中：

```

int area(int length, int width)      // 计算矩形面积
{
    if (length<=0 || width <=0) error("non-positive area() argument");
    return length*width;
}

```

上面程序实现了对于 `area()` 调用的所有错误处理。因此我们不再需要调用 `framed_area()`。进一步改进，我们可能需要对于错误信息的更准确描述。

函数的参数检查看上去很简单，但是为什么人们不总这么做呢？不注意错误处理是一个原因，粗心大意是另一个原因。此外，还有许多其他因素：

- 我们不能改变函数定义：在函数库内定义的函数因某种原因不能被改变。原因可能是该函数也被其他程序调用，相关的错误处理也可能不一致。该函数也可能为其他人所有，你没有相关源代码。该函数也可能属于某一个会定期更新的库，如果你修改了原函数，当库更新时，你不得不再次修订它。
- 被调函数不知道应该如何处理错误：这是库函数的典型应用。库的作者可以检测到错误，但是只有你才知道应该如何处理错误。
- 被调函数不知道它是在哪里被调用的：当你获得一个错误信息后，它会告诉你某个错误发生了，但不会告诉你程序是如何执行到这个点上的。但有时候，你会需要更详细地了解错误信息。
- 性能：对于小函数来说，错误检查的代价可能会超过计算本身。例如，对于 `area()` 函数来说，错误检查代价超过函数本身两倍（在这里，指的是需要执行的机器指令，而不是源代码长度）。对某些程序来说，这很重要。特别是当函数之间相互调用的时候，相关的参数变化不大，但是参数检查会被反复执行。

那我们应该怎么做呢？除非你有足够的理由，否则参数检查还是应该在函数内部完成。在介绍一些相关内容后，我们将在 5.10 节中继续讨论如何处理错误参数的问题。

5.5.3 报告错误

让我们考虑另外一个问题：在检查一系列参数后，一旦发现了一个错误，你应该如何做？有时，你可以返回一个“错误值”，例如：

```

// 要求用户输入 yes 或 no 作为应答
// 如果应答错误，返回 'b' (非 yes 或 no)
char ask_user(string question)
{

```

```
cout << question << "? (yes or no)\n";
string answer = " ";
cin >> answer;
if (answer == "y" || answer == "yes") return 'y';
if (answer == "n" || answer == "no") return 'n';
return 'b'; // 'b' 表示“错误应答”
}

// 计算矩形面积
// 返回 -1 表示参数错误
int area(int length, int width)
{
    if (length <= 0 || width <= 0) return -1;
    return length * width;
}
```

如上所示，我们可以让被调函数进行详细检查，同时让调用者按需要处理错误。看上去这种方法是可行的。但在某些情况下，这种方法会带来很多问题使得它实际上不能被接受：

- 所有调用者和被调函数都需要进行检查。调用者要进行的检查可能很简单，但还必须要编写这段代码，并决定在错误发生时候如何进行处理。
- 调用者可能会忘记做错误检查。这可能导致程序在运行时出现不可预测问题。
- 许多函数并没有可以用作标记错误信息的额外返回值。例如，一个用于从输入设备读入整数的函数（例如 `cin` 的操作符 `>>`）的返回值可以是任意整数。因此不能用一个专门的整数来表示错误信息。

对上面程序中的第二个例子，若调用者忘记了错误检查，这会导致某些不可预见问题。例如，

```
int f(int x, int y, int z)
{
    int area1 = area(x, y);
    if (area1 <= 0) error("non-positive area");
    int area2 = framed_area(1, z);
    int area3 = framed_area(y, z);
    double ratio = double(area1) / area3;
    ...
}
```

你看出错误在哪里了么？问题就是缺少了错误检查。因为没有明显的错误代码，这类错误往往很难被发现。

试一试

测试函数的不同输入和返回值。输出函数 `area1`、`area2`、`area3` 和 `ratio` 的值。尝试插入各种测试程序直到所有错误都被检测到。如何才能知道所有错误都被找到了呢？这不是一个脑筋急转弯问题，在本例中，你可以通过输入有效的参数检测所有的错误。

还有另外一种解决这一问题的方法：使用异常处理。

5.6 异常

与大多数现代编程语言类似，C++ 也提供了一种错误处理机制：异常。为了保证检测

到的错误不会被遗漏，异常处理的基本思想是把错误检测（在被调函数中完成）和错误处理（在主调函数中完成）分离。异常提供了一条可以把各种最好的错误处理方法组合在一起的途径。错误处理很繁琐，但异常可以让它变得简单一些。

 异常的基本思想是：如果一个函数发现一个自己不能处理的错误，它不是正常返回，而是抛出（throw）一个异常来表示错误的发生。任何一个直接或间接的函数调用者都可以捕捉到这一异常，并确定应该如何处理。函数可以用 try 语句（细节情况在下面章节中介绍）来处理异常：把所要处理的异常情况罗列在 catch 语句后。如果出现一个没有被任何调用函数处理的异常，程序终止运行。

在后续章节中（第 14 章），我们还将介绍异常的一些高级用法。

5.6.1 参数错误

下面是函数 area() 带异常处理的版本：

```
class Bad_area {};  
// 一个专门报告 area() 错误的类  
  
// 计算矩形面积  
// 在参数错误时抛出 Bad_area 异常  
int area(int length, int width)  
{  
    if (length<=0 || width<=0) throw Bad_area();  
    return length*width;  
}
```

如果参数正确，我们会返回计算的面积；否则结束函数 area()，并抛出异常，希望这个异常能够被捕获并做出相应错误处理。Bad_area 是一个我们定义的新类型。它的目的是作为函数 area() 中异常的标识，以便被捕获时能够确认异常来自哪里。用户自定义类型（类和枚举）将在第 9 章讨论。需要注意的是 Bad_area{} 表示“创建一个 Bad_area 类型的缺省值对象”。因此 throw Bad_area{} 表示“创建一个 Bad_area 类型的对象并抛出它”。

现在我们可以这样写：

```
int main()  
try {  
    int x = -1;  
    int y = 2;  
    int z = 4;  
    // ...  
    int area1 = area(x,y);  
    int area2 = framed_area(1,z);  
    int area3 = framed_area(y,z);  
    double ratio = area1/area3;  
}  
catch (Bad_area) {  
    cout << "Oops! bad arguments to area()\n";  
}
```

首先要注意的是，上面的错误处理针对的是所有对 area() 的调用，包括主函数里的一次调用和两个通过 framed_area() 的间接调用。其次，很明显，如何处理错误与检测错误是分离的：main() 不知道哪个函数做了 throw Bad_area{} 动作，area() 不知道哪个函数会捕捉它所抛出的 Bad_area 异常。对于使用了许多库的大程序来说，这一分离非常重要。因为在编程时没有人希望同时对应用程序和库代码进行修改，所以没有人能够“通过在正确位置简

单增加几行代码来修正错误”。

5.6.2 范围错误

大多数实际程序都需要处理数据集合，即，会用各种类型的数据元素的表格、列表等来完成某项任务。在 C++ 语言中，我们一般把“数据集合”称为容器（container）。最常用的标准库容器是 4.6 节中介绍的 `vector`。一个 `vector` 中包含了一组数据。我们可以通过 `vector` 的成员函数 `size()` 来获得数据的个数。如果我们引用了一个不在有效范围 `[0:v.size())` 内的下标的话，会出现什么情况呢？需要注意的是 `[low:high)` 表示从 `low` 到 `high-1` 的下标范围，它包括 `low` 但不包括 `high`：



在回答上面的问题前，我们先看看另外一个问题和它的答案：

“为什么要这么做呢？”毕竟，你应该明白下标只能在范围 `[0:v.size())` 内，因此确保如此就可以了呀！

话虽然是这么说的，但是实际上，很难保证这种情况不会发生。看看下面这个似乎合理的程序：

```
vector<int> v; // 一个整型 vector
for (int i; cin>>i;)
    v.push_back(i); // 获得值
for (int i = 0; i<=v.size(); ++i) // 打印值
    cout << "v[" << i << "] == " << v[i] << '\n';
```

你看出了问题了么？试着把它识别出来。这不是一个一般性的错误。这种错误往往是由我们自身的原因导致，特别是在工作到很晚我们很累的时候。当我们很劳累或者很毛躁的时候，这类错误经常会出现。在我们对 `v[i]` 进行操作时候，我们用 `0` 和 `size()` 来保证 `i` 总是在合法的范围内。

不幸的是，我们犯了一个错误。仔细看看 `for` 循环：它的终止条件是 `i<=v.size()` 而不是 `i<v.size()`。这会导致一个不幸的结果：如果我们读入了 5 个整数，它会试图输出 6 个结果。因为我们试图读 `v[5]` 的值，而它已经超出了 `vector` 的存储空间范围。这类错误是非常普遍的而且很“出名”，人们为它起了很多名字：偏一位错误（off-by-one error）；范围错误（range error），因为下标不在 `vector` 的合法值范围内；边界错误（bounds error），因为下标不在 `vector` 的合法边界内。⚠

为什么不用“范围 `for` 语句”来表示循环？这样做的话，循环的结尾就不会出错。但是，对这个循环来说，不仅需要每个元素的值，还需要它们的索引（下标）。范围 `for` 语句不能直接这么做。

下面是一个具有同样问题的简单版本：

```
vector<int> v(5);
int x = v[5];
```

然而，我们还是怀疑你没有认识到这个问题的真实性和严重性。

当我们犯了这个错误后，实际会发生什么情况呢？`vector` 的下标操作知道元素的个数，因此它可以检测是否出错（我们所使用的 `vector` 也是可以的，参见 4.6 节和 14.4 节）。如果

检查到错误，下标操作将抛出一个名为 `out_of_range` 的异常。因此，如果程序的代码有范围错误并导致了异常的话，我们至少能够捕捉到一个异常信息。

```
int main()
try {
    vector<int> v; // 一个整型 vector
    for (int x; cin >> x; )
        v.push_back(x); // 输入数值
    for (int i = 0; i <= v.size(); ++i) // 输出数值
        cout << "v[" << i << "] == " << v[i] << '\n';
} catch (out_of_range) {
    cerr << "Oops! Range error\n";
    return 1;
} catch (...) { // 捕获所有其他异常
    cerr << "Exception: something went wrong\n";
    return 2;
}
```

需要注意的是范围错误可以被认为是 5.5.2 节中提到的参数错误的一个特例。我们不能保证自己对 `vector` 下标的范围检查总是正确的，因此我们让 `vector` 的下标操作来做这一检查。正如上文所述，`vector` 的下标函数（名为 `vector::operator[]`）能够通过抛出异常来报告错误。它还能做什么吗？如果发生了一个范围错误的话，它是不知道我们将会如何处理的。`vector` 的作者甚至不知道 `vector` 是属于程序的哪一段代码。

5.6.3 输入错误

我们将把处理输入错误的细节讨论延后到 10.6 节。不过，一旦输入错误被发现，利用与处理参数错误和范围错误相同的技术，它将会被迅速处理。这里，我们只展示如何判断输入是否正确。下面是输入一个浮点数的情况：

```
double d = 0;
cin >> d;
```

通过测试 `cin`，我们可以确定最后一个输入操作是否成功：

```
if (cin) {
    // 输入成功，我们可以做下一次输入操作
}
else {
    // 输入操作失败，我们做一些错误处理
}
```

有几种原因可能会导致输入操作失败。其中一个原因就是 `>>` 操作输入的不是所要求的 `double` 类型数据。

在开发工作的早期，我们主要关注于发现错误，但并没有给出特别好的办法来解决它。我们做的仅仅是报告错误并终止程序。下面，我们将尝试更好的办法来处理它。例如：

```
double some_function()
{
    double d = 0;
    cin >> d;
    if (!cin) error("couldn't read a double in 'some_function()'!");
    // 做一些有益的事情
}
```

这里 `!cin`（“非 `cin`”，即 `cin` 处在有问题的状态）表示前一个 `cin` 的操作失败了。

传递给函数 `error()` 的字符串将被输出，它可以作为调试的有益帮助或者反馈给用户的信息。这个对于很多程序都很有用的功能 `error()` 应该如何编写呢？因为我们不知道应该如何处理返回值，所以这个函数没有返回值。它在输出信息后将直接终止程序。此外，在终止程序前，我们可以做一些次要的操作，例如保持窗口一段足够长时间以便我们阅读信息。显然，这是异常处理应该做的工作（参见 7.3 节）。

标准库定义了一些异常，例如 `vector` 的 `out_of_range`。此外，标准库还提供 `runtime_error` 异常。`runtime_error` 对我们非常有用，因为它包含一个字符串，可被错误处理函数使用。有了它，`error()` 可以被写成下面的形式：

```
void error(string s)
{
    throw runtime_error(s);
}
```

当我们想处理 `runtime_error` 的时候，捕捉到它就可以了。对于简单程序来说，在 `main()` 中捕捉 `runtime_error` 更理想：

```
int main()
try {
    // ... 我们的程序 ...
    return 0;      // 0 表示成功
}
catch (runtime_error& e) {
    cerr << "runtime error: " << e.what() << '\n';
    keep_window_open();
    return 1;      // 1 表示失败
}
```

函数 `e.what()` 将从 `runtime_error` 中提取错误信息。在下面语句

```
catch(runtime_error& e) {
```

中的 `&` 表示我们希望“以引用方式传递异常”。现在，我们暂时把它看作一种不相关的技术。在 8.5.4 ~ 8.5.6 节中，我们会详细解释通过引用传递参数的含义。

注意，这里我们使用 `cerr` 而不是 `cout` 进行错误信息输出：`cerr` 与 `cout` 用法相同，只是它是专门用于错误输出的。缺省情况下，`cerr` 和 `cout` 都输出到屏幕上。但是 `cerr` 没有经过优化更适合错误信息输出，在一些操作系统中它还可以被转向到其他输出目标，例如一个文件中。使用 `cerr` 也有助于我们编写与错误相关的文档。因此，我们使用 `cerr` 作为错误信息输出。

显然，`out_of_range` 与 `runtime_error` 不同。因此若代码捕获 `runtime_error` 异常，不会处理 `out_of_range` 错误（可能是 `vector` 或者其他标准库容器导致的）。但是，`out_of_range` 与 `runtime_error` 都是“异常”，我们可以对异常进行一些通用的处理：

```
int main()
try {
    // 我们的程序
    return 0;      // 0 表示成功
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    keep_window_open();
    return 1;      // 1 表示失败
}
```

```

catch (...) {
    cerr << "Oops: unknown exception!\n";
    keep_window_open();
    return 2;      // 2 表示失败
}

```

这里我们加上 `catch(...)` 来处理任何其他类型的异常。

我们用来同时处理 `out_of_range` 与 `runtime_error` 两种异常的是一个单一类型 `exception`, 它是两者的公共基类 (超类型, *supertype*)。这是一种非常有用的通用技术, 我们将在第 18 ~ 21 章中详细介绍。

需要注意的是 `main()` 的返回值传递给了调用程序的“系统”。一些系统 (例如 Unix) 经常会用到这些返回值, 而另一些系统 (例如 Windows) 会忽略它们。返回值为 0 表示 `main()` 成功完成, 而非 0 返回值表示某些错误发生了。

在使用 `error()` 的时候, 你可能希望能同时传递两部分信息来描述所发生的问题。在这种情况下, 我们可以把这两部分信息连接起来作为一个字符串传递。因此, 我们提供了第二个版本的 `error()`:

```

void error(string s1, string s2)
{
    throw runtime_error(s1+s2);
}

```

在我们的需求极大提高, 以及我们作为设计师和程序员的水平相应提高之前, 这种简单的错误处理方式就够用了。需要注意的是, `error()` 的使用与程序执行路径上有多少函数调用是无关的: `error()` 会查找距离最近的捕获 `runtime_error` 的 `catch` 操作, 通常就在 `main()` 中。使用异常和 `error()` 的例子, 可以参考 7.3 节和 7.7 节的内容。如果某个异常没有被捕获到, 缺省情况下, 你会得到一个系统错误 (“未捕获异常” 错误)。

试一试

尝试看看未捕获异常错误会是什么样: 运行一个使用了 `error()` 而没有捕获操作的小程序。

5.6.4 窄化错误

在 3.9.2 节中, 我们曾看过一个令人讨厌的错误: 当我们给一个变量赋了一个“太大”的值后, 这个值会被截断。例如:

```

int x = 2.9;
char c = 1066;

```

这里 `x` 的值是 2 而不是 2.9, 因为 `x` 是整型, 而整型没有小数部分, 只有整数部分 (这是显然的)。与之类似, 如果我们使用 ASCII 字符集, `c` 的值将是 42 (表示字符 *), 而不是 1066, 因为字符集中没有值为 1066 的字符。

在 3.9.2 节中, 我们已经看到如何通过测试来确保截断错误不发生。有了异常 (和模板, 参见 14.3 节), 我们可以编写函数来测试能够引起值改变的赋值或初始化操作, 有错误发生时, 抛出 `runtime_error` 异常。例如:

```

int x1 = narrow_cast<int>(2.9);           // 抛出异常
int x2 = narrow_cast<int>(2.0);           // 正常
char c1 = narrow_cast<char>(1066);        // 抛出异常
char c2 = narrow_cast<char>(85);          // 正常

```

这里 `<…>` 的用法与 `vector<int>` 中的尖括号相同。当需要确定一个类型而不是数值时，我们可以这样使用。它被称为模板参数（template parameter）。当需要进行类型转换，而不确定数值“是否适合”目标类型时，我们可以使用 `narrow_cast`。它是在 `std_lib_facilities.h` 中定义并用 `error()` 实现的。单词 `cast` 的意思是“类型转换”，它暗示进行转换的对象是有问题的（就像对一条断腿固定石膏模一样）。需要注意的是，类型转换并不会改变操作数，而是生成了一个所要求类型的新数值。

5.7 逻辑错误

在解决了开始的编译和链接错误后，程序就能运行了。通常情况下，此时的程序要么没有输出要么输出结果是错误的。产生这样结果的原因有很多。具体原因可能是你所理解的程序逻辑是错误的；可能你所编写的程序并不是你所设想的；可能你写控制语句时犯了一个“低级错误”；或者其他原因。通常，逻辑错误是最难被发现和排除的，因为在这种情况下计算机所做的正是你让它做的事情。你此时的任务是要发现你让计算机做的事情为什么没有反映你的真实意愿。基本上，我们可以认为计算机是一个速度非常快的笨蛋。它只是精确地完成你让它做的事情，这一点有时会让人感到很尴尬。

让我们通过一个简单的例子来理解逻辑错误。下面的代码在一组数据中找出最低、最高和平均温度：

```

int main()
{
    vector<double> temps;                      // 温度

    for (double temp; cin>>temp;)             // 读入温度值
        temps.push_back(temp);

    double sum = 0;
    double high_temp = 0;
    double low_temp = 0;

    for (int x : temps)
    {
        if(x > high_temp) high_temp = x;      // 找出最高
        if(x < low_temp) low_temp = x;        // 找出最低
        sum += x;                            // 温度求和
    }

    cout << "High temperature: " << high_temp << '\n';
    cout << "Low temperature: " << low_temp << '\n';
    cout << "Average temperature: " << sum/temps.size() << '\n';
}

```

为了测试上述程序，我们输入 2004 年 2 月 16 日德克萨斯州拉伯克（Lubbock）气象中心测得的每小时温度值（德州使用的是华氏温度）。

```

-16.5, -23.2, -24.0, -25.7, -26.1, -18.6, -9.7, -2.4,
 7.5, 12.6, 23.8, 25.3, 28.0, 34.8, 36.7, 41.5,
 40.3, 42.6, 39.7, 35.4, 12.6, 6.5, -3.7, -14.3

```

输出是：

```
High temperature: 42.6
Low temperature: -26.1
Average temperature: 9.3
```

初学者会认为上面的程序是没问题的。不负责的程序员会把它直接交付用户。谨慎的做法应该是用另外一组数据再次测试程序。这组数据来自 2004 年 7 月 23 日。

```
76.5, 73.5, 71.0, 73.6, 70.1, 73.5, 77.6, 85.3,
88.5, 91.7, 95.9, 99.2, 98.2, 100.6, 106.3, 112.4,
110.2, 103.6, 94.9, 91.7, 88.4, 85.2, 85.4, 87.7
```

这一次，输出结果是：

```
High temperature: 112.4
Low temperature: 0.0
Average temperature: 89.2
```

哦，一定是什么地方出问题了。7 月份的拉伯克出现严寒（0.0 华氏度大约是零下 18 摄氏度）将意味着世界末日！你能找出错误在哪里吗？原因在于 `low_temp` 的初值是 0.0，除非有一个温度值低于 0.0，否则它将一直是 0.0。

试一试

运行上面的程序。检验一下输入数据确实会产生那样的结果。尝试一下利用其他输入数据“打破”程序（即令程序输出错误结果）。看看你至少需要输入多少组数据，才会令程序出错。

不幸的是，程序中还有其他错误。如果所有温度值都低于 0 的话，会发生什么？`high_temp` 的初始化与 `low_temp` 存在同样的问题：除非有一个温度值高于 0.0，否则 `high_temp` 将始终是 0.0。在南极，这个程序同样会出问题。

这类错误是相当典型的。在程序编译时候它不会出错，对于“合理的”输入也不会出错。但是，我们忘记了仔细思考应该将哪些数据定义为“合理的”。这个程序的改进如下：

```
int main()
{
    double sum = 0;
    double high_temp = -1000;           // 初始化值低于可能值
    double low_temp = 1000;             // 初始化值高于可能值
    int no_of_temps = 0;

    for (double temp; cin>>temp;) {   // 读入温度
        ++no_of_temps;                 // 温度数据计数
        sum += temp;                  // 计算加总数据
        if (temp > high_temp) high_temp = temp; // 找出最高
        if (temp < low_temp) low_temp = temp; // 找出最低
    }

    cout << "High temperature: " << high_temp << '\n';
    cout << "Low temperature: " << low_temp << '\n';
    cout << "Average temperature: " << sum/no_of_temps << '\n';
}
```

这个程序正确么？你如何确定？你应该如何准确定义程序的“正确性”？哪里的温度

值会达到 1000 和 -1000 呢？想一想关于“魔术常量”的警告（5.5.1 节）。程序中使用 1000 和 -1000 这样的文字常量不是一种好的编程风格，但这两个值也会有问题么？是否有地方的温度会低于华氏 -1000 度（摄氏 -573 度）呢？是否有地方的温度会高于华氏 1000 度（摄氏 538 度）呢？

试一试

查阅一下相关资料，为我们程序的 `min_temp`（“最低温度”）和 `max_temp`（“最高温度”）设定合适的常量值。这些常量将决定我们程序的适用范围。

5.8 估计

想象一下，你已经写了一个进行简单计算的程序，例如计算六边形面积。运行这个程序你得到的结果是 -34.56。你知道它肯定有问题，为什么？因为面积不可能是负数。接下来，你找到并修改了相应错误（不管它是什么），然后再次得到一个结果 21.65685。这一次对了么？它很难说，因为我们一般不能马上说出计算六边形的公式。为了避免将一个产生可笑结果的程序交付用户而使我们出丑，我们应该在交付之前检查程序结果是否合理。在本例中，这个工作很简单。六边形与正方形很接近。在纸上画一个六边形，目测一下，它接近一个 3 乘 3 的正方形。这样一个正方形的面积是 9。真倒霉，结果 21.65685 不可能是对的。因此我们继续修改程序，新程序的计算结果 10.3923。这一次，它可能是正确的了！

这种方法与六边形无关，其关键思想是，除非我们知道正确的结果大概是什么，或者与什么比较类似，否则我们不能判定得到的结果是否合理。要不断问自己如下问题：

1. 这个问题的答案合理么？

还应该问一个更一般的（通常也更难）问题：

2. 我们应该如何判定一个结果是否合理？

这里，我们没有问“最准确的答案是什么？”或“正确答案是什么？”这是我们编写的程序要告诉我们的。我们所要知道的就是这个答案是合理的。只有确定了结果的合理性后，我们才能做下一步工作。

估计（estimation）是一种优雅的艺术，它将常识与一些用来解决常见问题的非常简单的数学方法结合起来。一些人很擅长在头脑中估计，但我们更提倡“在信封背面”随意写写画画，因为这种方法能够帮助我们减少错误。我们这里所说的估计是一种非正式的技术。有时候它会被（幽默地）称为瞎估计（guesstimation），因为它是将一点猜测和一点计算结合起来的方法。

试一试

我们的六边形的每边长都是 2 厘米。得到的结果是正确的吗？亲自动手“在信封背面”算一下。在纸上把它画出来，不要觉得这太简单了。许多著名的科学家都有这样一种令人敬佩的能力：用一支笔在信封背面（或餐巾纸上）估计出问题的近似结果。这是一种能力，实际也是一种简单的习惯，它能够帮助我们节省很多时间，减少很多错误。

通常，估计的过程包括对正确计算所需输入数据的估计，我们还未对此进行过讨论。假

定我们要测试一个估计城市间驾驶时间的程序。开车从纽约到丹佛花费 15 小时 33 分钟，这个时间是否合理呢？从伦敦到尼斯呢？为什么合理或者为什么不合理呢？在回答这些问题时，你需要用什么数据来估计行车时间呢？通常，在互联网上快速搜索一下是最有用的方法。例如，2000 英里是纽约和丹佛之间距离的合理估计。对于驾驶汽车来说，保持每小时 130 英里的平均速度是很困难的（或者是非法的）。因此 15 小时的结果是不合理的（ 15×130 略小于 2000）。你可以检验一下：我们既高估了距离，也高估了平均速度，但在检验合理性时，我们不需要非常准确，只要估计得差不多就可以了。



试一试

估计一下上述开车时间。同时，也估计一下相应的飞行时间（假定乘坐普通的民航航班）。然后，利用更准确的数据来验证你的估计，例如地图和时刻表。我们建议使用互联网资源。

5.9 调试

当你写完一个程序后（草稿？），它会有不少错误。小程序偶尔会一次通过。但如果一个复杂的大程序也出现这种情况的话，你一定要保持一个非常非常谨慎的态度。如果它确实第一次运行就正确的话，赶紧告诉你的朋友们来庆祝一下吧，因为这种事情不是每年都有的。

因此，当你写完代码后，你要做的就是找到并排除错误。这一过程称为调试（debugging），错误被称为 bug。据说 bug（有昆虫的意思）一词来自于早期真空管计算机时代，那时的计算机是占据了很大空间的真空管计算机，当小昆虫进入电路板后就会导致硬件故障。一些人将 bug 一词借用过来专指软件中的错误。其中最著名的就是 Grace Murray Hopper，COBOL 编程语言的发明人（参见 22.2.2.2 节）。这个词第一次出现已经是五十多年以前的事情了，现在它已经被人们普遍接受了。查找并排除错误的过程被称为调试。

调试可以被简单描述为：

1. 让程序编译通过。
2. 让程序正确链接。
3. 让程序完成我们希望它做的工作。

基本上，我们要一次次重复这个过程：对于大程序，需要上百次，上千次，年复一年。每当程序不能正常工作，我们就要找出问题并修正。我认为在编程中调试是最乏味、最费时间的工作，因此应该不遗余力地做好设计和编码工作，以使除错的时间降到最低。有的人在调试过程中体验到搜寻 bug 和深入程序精髓的快乐——调试可以和视频游戏一样让人上瘾，会把程序员没日没夜地黏在终端前（以个人的经验，我可以证实这一点）。



下面是调试时不要犯的错误：

```
while(程序出了问题){ // 伪代码
    在程序中随机地查找那些“看上去有问题”的代码并修订它们,
    使它们看上去更好一些
}
```

为什么我们要提到这些呢？显然，糟糕的方法会使成功的机会变得很低。不幸的是，上述行为恰好概括了许多人在深更半夜工作茫无头绪时所做的“无用功”。

调试中的关键问题是：

我如何知道程序是否真正运行正确呢？

如果你不能回答这个问题，你将会陷入长时间、乏味的调试工作中，而且你的用户也很可能陷入麻烦。我们会反复强调这个问题，任何有助于回答此问题的信息都会减少调试工作量，并有助于生成正确、可维护的程序。实际上，我们宁愿程序设计良好，使得错误无处藏身。一般来说这个目标很难达到，但我们还是会在程序设计上下功夫，以最小化出错的几率，同时最大化发现错误的几率。

5.9.1 实用调试建议

在编写代码前一定要仔细考虑调试问题。如果已经写完很多行代码之后你才想到应该如何简化调试问题的话，那就太晚了。

首先你要决定如何报告错误：“使用 `error()` 并在 `main()` 中捕获异常”应该是你从本书学到的一个标准答案。

要提高程序的易读性，这样你会有更多机会发现错误所在：

- 为代码做好注释。这并不意味着“加上大量注释”。能靠代码本身表达清楚的，不要用注释。注释的内容应该是你不能在代码里说清楚的部分。你应该用尽量简洁、清楚的语言把它们说清楚，包括：
 - 程序的名称；
 - 程序的目的；
 - 谁在什么时候写了这个代码；
 - 版本号；
 - 复杂代码片段的目的是什么；
 - 总体设计思想是什么；
 - 源代码是如何组织的；
 - 输入数据的前提假设是什么；
 - 还缺少哪一部分代码，程序还不能处理哪些情况。
- 使用有意义的名字。
 - 这并不意味着使用“长名字”。
- 使用一致的代码层次结构。
 - 你是代码的负责人，集成开发环境可以帮助但不能替代你做所有事情。
 - 本书所使用的编程风格可以作为一个有益的起点。
- 代码应该被分成许多小函数，每个函数表达一个逻辑功能。
 - 尽量避免超过一或两页的函数；大多数函数应该很短。
- 避免使用复杂的程序语句。
 - 尽量避免使用嵌套的循环，嵌套的 `if` 语句，复杂的条件等。不幸的是，有时你必须这样做，但请记住复杂代码是错误最容易隐藏的地方。
- 在可能的情况下，使用标准库而不是你自己的代码。
 - 同样是完成某个功能，标准库一般会比你自己写的程序考虑得更周全，经过了更完备的测试。

上面的描述有些抽象，但在后续的篇幅中，我们会通过一个个的例子来向你详细解释。程序首先要编译通过。在这个阶段，编译器显然是你最好的助手。它给出的错误信息

通常是很有用的，虽然我们总希望得到更准确的信息。除非你是一个真正的专家，否则你还是假定编译器是正确的为好。当然，如果你是一个真正的专家，这本书也不是为你写的。有时，你会觉得编译器遵循的规则实在是太愚蠢而且没有必要（通常并非如此），这些规则能够而且应该更简单（的确，但是它们不是这样的）。然而，俗话说“糟糕的工匠才会抱怨他的工具”。好的工匠了解自己工具的长处和短处，并能在此基础上对自己工作进行相应调整。下面是一些常见的编译错误：

- 每一个字符串常量都用双引号终止了吗？

```
cout << "Hello, << name << '\n';      // 错误
```

- 每一个字符常量都用单引号终止了吗？

```
cout << "Hello, " << name << '\n';      // 错误
```

- 每一个程序块都终止了吗？

```
int f(int a)
{
    if (a>0) /* 完成相关工作 */ else /* 完成另一部分工作 */
}      // 错误
```

- 所有圆括号都一对一匹配吗？

```
if (a<=0      // 错误
    x = f(y);
```

编译器一般会“稍晚些”报告这类错误，因为它不知道你本来想在 0 之后输入一个右括号。

- 每一个名字都声明了么？

- 你是否包含了所有必需的头文件（目前，应该用 #include "std_lib_facilities.h"）？
- 所有名字都在使用之前进行声明了么？
- 你是否正确拼写了所有名字？

```
int count; /* ... */ ++Count;      // 错误
char ch;   /* ... */ Cin>>c;      // 两个错误
```

- 你用分号终止了每个表达式语句吗？

```
x = sqrt(y)+2 // 错误
z = x+3;
```

在本章的简单练习中，我们将给出更多的例子。同时，请记住 5.2 节中关于错误的分类。

在程序的编译和链接完成后，下一步就是最难的部分了：找出程序没有按照我们的意图去运行的原因。你要检查输出结果，并尽力搞清楚为什么程序会产生这样的结果。实际中，通常首先你会看着一个黑屏（或窗口），奇怪于你的程序没有输出任何结果。对于 Windows 控制台程序，通常面临的第一个问题就是，在你能够看清楚输出结果之前（如果有的话），控制台窗口就消失了。一种解决办法是在 main() 的末尾调用 std_lib_facilities.h 中的 keep_window_open()。这样，窗口在退出前会等待一个输入。在给出一个输入关闭窗口之前，你就可以查看输出结果了。

在查找错误时，你要从程序中最后一个能够确认正确的语句开始，逐条语句仔细检查。就好像你就是计算机在执行这个程序。程序的输出是否与你的预计相同呢？当然是不会了，

如果是的话，你就不用调试了。

- 通常，当你没有找到问题的时候，原因是你只“看到”了自己希望看到的，而不是你编写的程序，例如：

```
for (int i = 0; i<=max; ++i) {           // 糟糕（有两个错误）
    for (int i=0; 0<max; ++i);          // 打印 v 中的元素
        cout << "v[" << i << "]==" << v[i] << '\n';
    // ...
}
```

这个例子来自一个有丰富经验的程序员写的实际程序（我们认为它应该是在深夜编写的）。

- 通常，如果你没有找到问题所在，原因可能是在上一个你能确认的正确的输出和下一个输出（或没有输出）之间，代码太多。大多数编程环境都提供逐条语句执行程序的功能（“单步执行”）。最终，你肯定能学会使用这些工具。但对于简单问题和简单程序来说，你可以通过在程序中临时加入一些额外的输出语句（利用 `cerr`），来帮助你检查运行结果。例如：

```
int my_fct(int a, double d)
{
    int res = 0;
    cerr << "my_fct(" << a << "," << d << ")\n";
    // ... 执行结果有问题的代码 ...
    cerr << "my_fct() returns " << res << '\n';
    return res;
}
```

- 在可能会隐藏错误的语句中，加入检查不变式（即永远成立的条件，参见 9.4.3 节）的语句，例如：

```
int my_complicated_function(int a, int b, int c)
// 参数为正整数且 a<b<c
{
    if (!(0<a && a<b && b<c)) // ! 表示“非”，&& 表示“与”
        error("bad arguments for mcf");
    // ...
}
```

- 如果还是没有效果的话，在看起来不会有错的代码段中插入检查不变式的语句——如果你找不到错误，几乎可以肯定你是找错地方了。

陈述一个不变式的语句称为断言（`assertion` 或 `assert`）。

更有趣的是，还存在许多其他有效的调试方法。不同的人所使用的技术可能差别极大。 调试技术的差异中，有很多来自于要调试的程序的差异；另外一些则是源自于人们不同的思维方式。据我们所知，目前还没有一种最好的调试方法。但是有一件事要始终牢记：混乱的代码总是容易隐藏错误。尽你所能保证代码的简洁、有逻辑性和格式的工整吧，这将会大大减少你的调试时间。

5.10 前置条件和后置条件

现在，让我们回到前面的问题：如何处理函数的参数错误。基本上，函数调用是思考正确代码和捕捉错误的最佳位置：逻辑上，函数是一个独立计算的开始（函数返回是结束）。看看我们利用前一节介绍的调试技巧做了什么：

```

int my_complicated_function(int a, int b, int c)
// 参数为正整数且 a<b<c
{
    if (!(0 < a && a < b && b < c)) // ! 表示“非”，&& 表示“与”
        error("bad arguments for mcf");
    ...
}

```

首先，我们（在注释中）说明了函数对于参数的要求，然后在函数开始处检查这一要求是否满足（如果不满足则抛出一个异常）。

这是一个很好的基本策略。函数对于自己参数的要求被称为前置条件（precondition）：如果函数正确执行的话，这一条件必须为真。现在的问题是如果前置条件被违反的话（为假），那么我们应该怎么做。我们可以有两个选择：

1. 忽略它（希望 / 假设调用者会使用正确的参数）。
2. 检查它（并以某种形式报告错误）。

我们可以使用参数类型机制，它能让编译器进行最简单的前置条件检查，并在编译时报告错误。例如：

```
int x = my_complicated_function(1, 2, "horsefeathers");
```

这里，编译器会检查出“第三个参数应是整型”的函数要求（“前置条件”）被违反了。基本上，我们在本节中要讨论的是如何处理编译器不能检查的函数要求 / 前置条件。

 我们的建议是前置条件一定要在注释里说明（这样调用者可以知道函数的要求）。一个没有注释文档的函数会被认为能够处理每种可能的参数值。但是我们能够确信调用者会读这些注释并遵守其中的规则么？有些时候我们不得不相信调用者会这样做，但我们通常还是要遵循“被调用者进行参数检查”这一原则，它可以解释为“让函数检查自己的前置条件”。在没有其他原因的情况下，我们要坚持做到这一点。不做前置条件检查的常见理由包括：

- 没人会使用错误参数。
- 做前置条件检查会使我的程序变慢。
- 检查工作太复杂了。

当我们知道“谁”将调用这个函数的时候，第一个理由是很合理的，但在实际编程工作中，这一条件很难满足。

第二个理由成立的情况远比人们通常认为的要少得多。大多数情况下，它应该作为“过早优化”的例子被摒弃掉。如果前置条件检查被证实真的是程序的负担的话，你当然可以把它去掉。但是它所保证的程序正确性就不是那么容易能获得了，本来是它能捕获的一些错误，又需要你花费许多不眠之夜来查找了。

第三个原因是严重的。很容易（特别是对有经验的程序员来说）找到这种例子：前置条件检查所做的工作比执行函数本身还要多得多。一个例子就是字典查询操作：前置条件是字典是已排序的，而验证一个字典已排序的代价要远远高于单纯的查询操作。有时候，编写前置条件检查代码以及判定这部分代码是否正确是很困难的。但是，每当你编写函数的时候，你都应该考虑是否可以编写一个快速的前置条件检查代码。除非你有足够的理由，否则始终应该为函数编写前置条件检查代码。

编写前置条件（即使是以注释形式）还可以显著地提高你的程序质量：它能强迫你考虑函数的需求是什么。如果你不能用几行注释把它简单、准确地描述出来的话，你可能还没有

真正地理解你要做什么。经验表明，编写前置条件注释和前置条件检查代码有助于你避免许多设计上的错误。我们说过讨厌调试工作；使用前置条件将有助于避免设计错误和及早发现使用错误。例如：

```
int my_complicated_function(int a, int b, int c)
// 参数为正整数且 a<b<c
{
    if (!(0 < a && a < b && b < c)) // ! 表示“非”，&& 表示“与”
        error("bad arguments for mcf");
    // ...
}
```

与下面的简化版本相比，上面的代码将会节省你的时间。

```
int my_complicated_function(int a, int b, int c)
{
    // ...
}
```

5.10.1 后置条件

使用前置条件将有助于我们避免设计错误和及早发现使用错误。这种显式说明需求的思想能够被应用在其他方面么？是的，你可以马上联想到：返回值！毕竟，我们一般都需要声明函数的返回值是什么。也就是说，如果一个函数返回一个值的话，我们总是要约定这个返回值是怎样的（否则的话调用者如何知道得到的是什么呢？）。让我们再次看一下面积函数（来自 5.6.1 节）：

```
// 计算矩形面积
// 如果参数错误的话，抛出一个 Bad_area 异常
int area(int length, int width)
{
    if (length <= 0 || width <= 0) throw Bad_area();
    return length*width;
}
```

这个程序检查了前置条件，但是它没有在注释里面对此进行说明（对于这么一个小函数来说，这还是可以接受的），而且假定计算是正确的（这种简单计算应该也没问题）。但是，我们可以更明确一些：

```
int area(int length, int width)
// 计算矩形面积
// 前置条件：长度和宽度是正数
// 后置条件：返回值是正数，表示矩形面积
{
    if (length <= 0 || width <= 0) error("area() pre-condition");
    int a = length*width;
    if (a <= 0) error("area() post-condition");
    return a;
}
```

我们不可能对后置条件进行完全检查，但是我们至少可以检查其中一部分：返回值是否是正数。

试一试

尝试找出一组数据，它们能够满足当前版本面积函数的前置条件，但不满足后置条件。

前置和后置条件提供了基本的程序完整性检查。从这个角度看，它们与不变式（9.4.3 节）、正确性（4.2 节和 5.2 节）和测试（第 26 章）等概念紧密相关。

5.11 测试

我们如何知道应该什么时候停止调试呢？不错，在找到所有错误前，我们应该继续调试，或者尽力去做。我们如何知道已经找到了最后一个错误了呢？答案是没有办法。“最后一个错误”是程序员之间的笑话：这种东西是不存在的。对一个大程序来说，我们永远不会找到“最后一个错误”。因为在寻找错误的同时，我们还要忙于按照新需求修改程序。

 除了调试以外，我们还需要一种系统地查找错误的方法。这被称为测试（test），我们将在 7.3 节、第 10 章的习题和第 26 章中详细介绍相关内容。基本上，测试是把一个巨大的、有系统地选择过的数据集输入给程序，然后把相关的结果与期望值进行比较。基于一组给定输入的一次程序运行被称为一个测试用例（test case）。实际程序可能会需要上百万个测试用例来进行测试。基本上，系统测试不可能靠人手工输入一个个测试用例。在介绍过后续几章内容并掌握了必要的工具后，我们再正式讨论测试。在此期间，我们需要谨记的是找到错误才是好的，这才是进行测试需要秉承的态度。看看下面的内容：

态度 1：我比任何程序都聪明！我将击败这些 @#\$%^ 代码。

态度 2：这部分代码我已经打磨了两周时间了。它是完美的。

你认为谁会找出更多的错误？当然，最好的情况是一个有经验的人带着一点“态度 1”，沉着、冷静、耐心地对程序中所有可能出错的地方进行系统地检查。好的测试人员是非常难能可贵的。

我们会尽量系统地选择测试用例，一般会包括正确和不正确的输入数据。7.3 节中会给出第一个示例。

简单练习

接下来会有 25 个代码片段，每一个都要被插入到下面这个“框架”中：

```
#include "std_lib_facilities.h"

int main()
try {
    <<your code here>>
    keep_window_open();
    return 0;
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    keep_window_open();
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    keep_window_open();
    return 2;
}
```

每段代码都有 0 个或多个错误。你的任务是找出并排除每个程序中的错误。当你排除了所有错误后，得到的程序编译、运行后将会输出“Success!”。即使你认为已经找到了一

个错误，你仍然需要输入（原始、未修改的）程序并测试它；因为你可能猜错了，或者程序中还有其他错误。这个练习的另一个目的是让你感受一下编译器对不同错误的反应是什么样的。你不需要输入上面的程序框架 25 次，用剪切、粘贴或类似的技术就可以了。不要通过删除一条语句来逃避问题，你应该试着用修改、增加或删除一些字符来排除问题。

```
1. Cout << "Success!\n";
2. cout << "Success!\n;
3. cout << "Success" << !\n"
4. cout << success << '\n';
5. string res = 7; vector<int> v(10); v[5] = res; cout << "Success!\n";
6. vector<int> v(10); v(5) = 7; if (v(5)==7) cout << "Success!\n";
7. if (cond) cout << "Success!\n"; else cout << "Fail!\n";
8. bool c = false; if (c) cout << "Success!\n"; else cout << "Fail!\n";
9. string s = "ape"; boo c = "fool"<s; if (c) cout << "Success!\n";
10. string s = "ape"; if (s=="fool") cout << "Success!\n";
11. string s = "ape"; if (s=="fool") cout < "Success!\n";
12. string s = "ape"; if (s+"fool") cout < "Success!\n";
13. vector<char> v(5); for (int i=0; 0<v.size(); ++i) ; cout << "Success!\n";
14. vector<char> v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";
15. string s = "Success!\n"; for (int i=0; i<6; ++i) cout << s[i];
16. if (true) then cout << "Success!\n"; else cout << "Fail!\n";
17. int x = 2000; char c = x; if (c==2000) cout << "Success!\n";
18. string s = "Success!\n"; for (int i=0; i<10; ++i) cout << s[i];
19. vector v(5); for (int i=0; i<=v.size(); ++i) ; cout << "Success!\n";
20. int i=0; int j = 9; while (i<10) ++j; if (j<i) cout << "Success!\n";
21. int x = 2; double d = 5/(x-2); if (d==2*x+0.5) cout << "Success!\n";
22. string<char> s = "Success!\n"; for (int i=0; i<=10; ++i) cout << s[i];
23. int i=0; while (i<10) ++j; if (j<i) cout << "Success!\n";
24. int x = 4; double d = 5/(x-2); if (d=2*x+0.5) cout << "Success!\n";
25. cin << "Success!\n";
```

思考题

1. 举出四种主要错误类型并给出它们的简洁定义。
2. 在学生练习程序中，什么类型的错误我们可以忽略？
3. 每一个完整的程序应该能提供什么保证？
4. 举出三种可以减少程序错误，开发出符合要求的软件的方法。
5. 为什么我们会讨厌调试？
6. 什么是语法错误？给出五个例子。
7. 什么是类型错误？给出五个例子。
8. 什么是链接时错误？给出三个例子。
9. 什么是逻辑错误？给出三个例子。
10. 列出四种本章中讨论的可能导致程序错误的因素。
11. 你如何能知道一个结果是合理的？回答这类问题，你会用到什么技术？
12. 对比一下由函数调用者来处理运行时错误和由被调函数来处理运行时错误的异同。
13. 为什么说使用异常比返回一个“错误值”要好？
14. 你应该如何测试一个输入操作是否成功？
15. 描述一下抛出和捕获异常的过程。
16. 有一个名为 v 的 vector，为什么 v[v.size()] 会导致范围错误？这一调用会导致什么结果？

17. 描述一下前置条件和后置条件的定义；并举个例子（不能用本章中的 area() 函数），最好是一个带有循环的计算过程。
18. 什么时候你不会测试前置条件？
19. 什么时候你不会测试后置条件？
20. 调试程序时的单步执行是指什么？
21. 在调试程序时，注释会有什么帮助？
22. 测试与调试有什么不同？

术语

argument error (参数错误)	logic error (逻辑错误)
assertion (断言)	post-condition (后置条件)
catch	pre-condition (前置条件)
compile-time error (编译时错误)	range error (范围错误)
container (容器)	requirement (需求)
debugging (调试)	run-time error (运行时错误)
error (错误)	syntax error (语法错误)
exception (异常)	testing (测试)
invariant (不变式)	throw
link-time error (链接时错误)	type error (类型错误)

习题

1. 如果你还没有完成本章中的“试一试”，请先完成相关练习。
2. 下面的程序获得摄氏温度值并将其转化为绝对温度。但这些代码有很多错误。找到这些错误，指出并修改它们。

```
double ctok(double c)      // 摄氏温度到绝对温度转换
{
    int k = c + 273.15;
    return int
}
int main()
{
    double c = 0;          // 声明输入变量
    cin >> d;             // 输入温度值，存入输入变量
    double k = ctok("c"); // 转换温度
    Cout << k << '\n';   // 输出温度
}
```

3. 绝对零度是能够达到的最低温度；它是 -273.15 摄氏度或 0K。即使上面的程序是正确的，当输入一个低于这个值的温度时，程序也应输出错误结果。检查一下，当输入一个低于 -273.15 摄氏度的数值时，主程序是否产生错误。
4. 重做习题 3，但这次把错误处理放在 ctok() 中。
5. 给这个程序增加一些功能，使它也可以把绝对温度转化为摄氏温度。
6. 编写一个程序，它可以实现摄氏温度转化为华氏温度和华氏温度转换为摄氏温度（公式见 4.3.3 节）。用估计的方法（5.8 节）看看你的结果是否合理。

7. 一元二次方程的形式如下

$$ax^2 + bx + c = 0$$

解这个方程，用到求根公式：

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

这里面有一个问题：如果 $b^2 - 4ac$ 小于零的话，它将出错。编写一个可以解一元二次方程的程序。建立一个可以计算一元二次方程根的函数，给定 a, b, c ，如果 $b^2 - 4ac$ 小于 0 就抛出一个异常。让程序的主函数调用这个函数，如果有错误由主函数捕获异常。当程序发现方程没有实根的时候，输出相应的信息。你如何确定程序的结果是合理的？你能检验结果的正确性么？

8. 编写一个程序，它能够读取并存储一列整数，然后计算前 N 个整数的和。首先提示用户输入 N ，然后读入一列整型值并存储在一个 `vector` 中，然后计算前 N 个值的和。例如：

“Please enter the number of values you want to sum:”

3

“Please enter some integers (press '| to stop):”

12 23 13 24 15 |

“The sum of the first 3 numbers (12 23 13) is 48.”

处理所有可能的输入。例如，如果用户要求加总数字的数目超过 `vector` 中的个数，输出错误信息。

9. 修改习题 8 的程序：如果结果不能表示为整型的话，输出一个错误信息。
10. 修改习题 8 的程序：使用 `double` 而不是 `int`。再创建一个 `double` 值的 `vector`，包含 $N-1$ 个相邻元素的差，并输出这个差 `vector` 的各个元素值。
11. 编写程序，输出尽量长的斐波那契序列，也就是说，序列的开始是 1 1 2 3 5 8 13 21 34。序列的后一个数是前两个数之和。找出 `int` 能允许的最大的斐波那契数。
12. 实现一个名为“公牛和母牛”（这不知道是谁命名的）的竞猜程序。程序用一个 `vector` 存储四个 0 到 9 之间的整数（如 1234，但不是 1122）。用户的任务是通过重复猜测找到这些数。例如，如果存储的是 1234 而用户猜测的是 1359 的话，程序的反馈结果是“1 头公牛 1 头母牛”。因为用户的猜测中有一个数字是正确的（1）并且它在正确的位置（1 头公牛），有一个数字是正确的（3）但它在错误的位置（1 头母牛）。反复这一猜测过程，直到用户找到四个公牛，即找到这四个数字且都在正确的位置。
13. 上一题的程序有点乏味，因为答案硬编码到程序中了。编写一个新版本：用户可以重复玩这个游戏（不需要停止或重启游戏），每一次游戏生成一组新的四个数的集合。你可以通过四次调用 `std::lib_facilities.h` 中的随机数发生器 `randint(10)` 来生成四个随机数。但你会发现，当你重复执行这个程序的时候，每一次程序都会给出四个相同的数。为了避免这一点，你可以要求用户输入一个数（可以是任意数字），然后在调用 `randint(10)` 之前先调用 `srand(n)`，这里 n 是用户所输入的数。这个 n 被称为种子（seed），不同的种子会产生不同的随机数序列。
14. 从标准输入中读入（星期几，数值）对，例如：

Tuesday 23 Friday 56 Tuesday -3 Thursday 99

将每一天对应的所有数值存入一个 `vector<int>` 中。输出这 7 个 `vector` 中的值。打印每个 `vector` 中的加总值。忽略输入中的非法日期，例如 `Funday`，但是接受同义词，例如 `Mon` 和 `monday`。输出被拒绝数值的个数。

附言

你认为我们过分强调错误了么？作为初学者我们可能会这么想。显然，一个自然的反应是“这么简单的程序不可能出错”。但是，错误总是会发生的。许多世界上最聪明的头脑都惊讶和困惑于编写正确程序所具有的难度。据我们的经验，好的数学家是最可能低估错误问题的人群。**△** 但我们会认识到：所编写的程序一次通过是一件超出我们能力范围的事。一定要引起重视！尽管我们会尽自己所能，但是错误不可避免地会出现在刚刚编写完的程序中。幸运的是，经过了 50 年或更长的时间，我们已经拥有了许多如何组织代码来最小化错误数目经验，以及相关的错误查找技术。本章中介绍的技术和示例就是一个好的开端。

编写一个程序

程序设计就是问题理解。

——Kristen Nygaard

编写程序需要不断地细化所实现的功能及其表达方式。接下来的这两章详细讲述了程序的开发过程，从一个并不十分清晰的想法开始，经过分析、设计、实现、测试、再设计和再实现等步骤，最终实现预期目标程序。本章主要讨论程序结构、用户自定义类型和输入处理等内容，目的是帮助读者了解在编写代码过程中如何去思考。

6.1 一个问题

程序的编写往往都是从一个问题出发，也就是说，借助程序来解决一个实际问题，因此正确理解问题对程序实现是非常关键的。毕竟，解决一个理解错误的问题的程序很可能是没有用处的，即使它是一个完美的程序。或许这个程序恰好对从来没有想到的某些问题是有效的，但这种幸运事件发生的概率非常小。因此，所设计的程序应该简单、清晰地解决要处理的问题。

在这个阶段，一个好的程序应该具有以下几个特点：

- 阐明设计和编程技术；
- 易于探究程序员做出的各种各样的决策及其相关考虑；
- 不需要很多新的语言结构；
- 对设计的考虑足够全面；
- 易于对解决方案进行改变；
- 解决一个易于理解的问题；
- 解决一个有价值的问题；
- 具有一个足够小，从而可完整实现、彻底理解的求解方案。

我们编写一个简单的计算器，实现计算机对输入表达式的常规算术运算。无疑这类程序是很有用的，在每个台式机中都安装有这样的计算器程序，甚至我们可以购买专门运行该程序的计算设备：袖珍计算器。

例如：输入

2+3.1*4

程序应该输出：

14.4

不幸的是，这样的计算器程序在我们的电脑上已经随处可见，它不会给我们带来任何新功能，但作为第一个程序，这已经足够了。

6.2 对问题的思考

我们如何开始？大体上说，我们要做的就是对问题和问题求解方法进行思考。首先，考

虑程序应该完成什么，人机交互的方式是怎样的。然后，考虑如何设计程序才能实现这样的功能。试着写出每个解决方案的简单框架，并检验它们的正确性。或许与朋友讨论这个问题及其解决方法，试着给朋友讲述你的想法，是一种很好地发现错误的方式，甚至比写下来都要好很多，因为纸（或者计算机）不能对你的假设提出疑问，不能反驳你的错误观点。总之，设计不是一个孤独的过程。

不幸的是，不存在对所有人和所有问题都有效的通用解决策略。有些书通篇都在帮助读者学习更好地求解问题，其他大部分书籍则侧重于程序设计。我们并不那么做，相反，本书针对一些个人能够处理的小规模问题给出若干有价值的通用求解策略，然后以微型计算器程序为例对这些策略进行验证。

建议读者带着疑问阅读关于计算器程序的讨论。实际上，一个程序的开发要经过一系列版本，每个版本实现了我们得到的一些推论。很明显，某些推论是不完全的甚至是错误的，否则可以更早就结束本章。随着讨论的深入，我们逐步给出了各种各样的关注点和推论的实例，这些都是设计者和程序员一直要面对和处理的，直到下一章结尾才完成这个程序的最终版本。

学习本章和下一章时要记住，实现程序最终版本的过程——提出部分解，产生想法和发现错误的历程——与程序最终版本本身同样重要，甚至比实现过程中碰到的语言技术细节更重要（后面还会讲到这些问题）。

6.2.1 程序设计的几个阶段

下面是程序开发涉及的几个术语。解决一个问题需要反复经历以下阶段：



- 分析 (Analysis): 判断应该做什么并且给出对当前问题理解的描述，称为需求集合 (a set of requirements) 或者规范 (specification)。我们并不详细讨论如何开发和撰写这些规范，这已经超出本书的范围，但问题的规模越大，这种规范就越重要。
- 设计 (Design): 给出系统的整体结构图，并确定具体的实现内容以及它们之间的相互联系。作为设计的一个重要方面，要考虑哪些工具（如函数库）有助于实现程序的结构。
- 实现 (Implementation): 编写代码、调试并测试，确保程序完成预期的功能。

6.2.2 策略



下面是一些对很多程序设计项目都有帮助的建议：

- 要解决的问题是什么？首要的事情是将要完成的目标具体化，包括建立问题的描述，或者分析已有描述的真实意图。这个时候应该站在用户而不是程序员的角度上，也就是说，应该考虑程序要实现什么功能，而不是怎样实现这些功能。例如：这个程序能够实现什么功能？用户与程序以什么方式进行交互？记住，大多数人都具有很丰富的计算机使用经验。
- 问题定义清楚吗？事实上，我们无法十分清晰地定义一个现实问题，即使是一个学生习题，也很难将其准确和具体地定义。但是，解决一个错误的问题是很遗憾的，所以必须弄清楚所要解决的问题是什么。另一个易犯的错误是我们容易把问题复杂化，在描述一个要处理的问题时总是表现得过于贪心 / 有野心。实际上，更好的方式是将问题简化，使程序易于定义、理解、使用和实现。一旦程序能够实现预期的功能，基于已有的经验可以实现它的第 2 版。

- 看上去问题是可处理的，但时间、技巧和工具是否足够？从事一项不可能完成的项目是没有意义的。因此，如果没有足够的时间来实现（包括测试）一个程序，最好不要启动这个项目。否则，需要获取更多的资源（包括时间）或者修改需求来简化任务。
- 将程序划分为可分别处理的多个部分。为了解决一个实际问题，即使是一个最小的程序也能够进一步细分。
 - 你知道有哪些工具、函数库或者其他能借助的东西？答案是肯定的，因为即使是学习编程的最初阶段，你也能使用 C++ 标准函数库的部分内容。以后，还会慢慢学习如何使用标准函数库的更多功能，还会用到图形和 GUI 库、矩阵库等。在获得了一些编程经验之后，通过简单的网络搜索就能发现更多的函数库。记住：在编写真正实用的软件时，重新设计基本模块是没有价值的。在学习编程的时候是另外一回事，通过重新设计基本模块可以更清楚地了解其实现过程。通过使用函数库节约的时间可以用于解决问题的其他部分，或者休息。但是，如何知道一个函数库适合于目前的任务，或者程序性能是否满足要求是一个很困难的问题。一种解决方法是咨询同事、讨论组，或者在使用函数库前首先使用例子进行验证。
 - 寻找可以独立描述的部分解决方案（或许能用在程序中的多个地方，甚至能用于其他程序）。发现这样的解决方案需要经验，因此本书提供了很多实例，目前我们已经用到了 `vector`、`string` 和 `iostream` (`cin` 和 `cout`)。本章给出第一个完整的实例，展示了用户自定义类型 (`Token` 和 `Token_stream`) 的设计、实现和使用。第 8 章，第 18 至 20 章给出了更多的实例，并给出了它们的设计思路。现在考虑一个类似的问题：如果要设计一辆汽车，首先应该确定它的每个组成部分，包括车轮、发动机、座椅、门把手等，在组装整车之前这些组件都可以独立生产。一辆汽车包含成千上万的组件，一个程序也是如此，只不过它的每个组件是代码而已。如果没有钢材、塑料和木材等原材料，我们是不能直接制造出汽车的，同样，没有语言提供的表达式、语句和类型，我们也不能直接编写出程序。设计和实现这种组件是本书的一个重要主题，也是软件开发的重要方法；参见第 9 章的用户自定义类型，第 19 章的类的层次结构和第 15 章的泛型。
- 实现一个小小的、有限的程序来解决问题的关键部分。当我们开始程序设计时，对要解决的问题并不十分了解。我们常常认为我们很了解（难道还不知道一个计算器程序是什么吗？），但实际上并不是这样。只有充分思考（分析）并且实验（设计和实现）之后才能深入理解要解决的问题，才能编写出一个好的程序。因此，实现一个小小的、有限的程序：
 - 引出我们的理解、思想和工具中存在的问题；
 - 看看能否改变问题描述的一些细节使其更加容易处理。当我们分析问题并给出初步设计时，预先估计出所有问题是几乎不可能的。我们必须充分利用代码编写和测试过程中的反馈信息。

有时这样一个用于实验的小程序称为原型（prototype）。如果第一个程序不能工作或者很难在此基础上继续下去，可以将其丢弃，基于已有的经验重新设计一个原型程序，直到找到一个满意的版本为止。不要在一个混乱的版本上继续，否则将会越来越混乱。

- 实现一个完整的解决方案，最好是能够利用最初版本中的组件。理想情况是逐步构建组件来编写一个程序，而不是一下子写出所有代码。要不然，你就得希望奇迹发生了，期待一些未经检验的想法能够实现我们设想的功能。

6.3 回到计算器问题

如何与计算器进行交互？这个问题容易解决：因为我们知道如何使用 `cin` 和 `cout`。但图形用户界面（GUI）直到第 21 章才讲述，因此这里使用键盘和控制台窗口。假设从键盘输入表达式，然后计算并将结果显示在屏幕上。例如：

```
Expression: 2+2
Result: 4
Expression: 2+2*3
Result: 8
Expression: 2+3-25/5
Result: 0
```

$2+2$ 和 $2+2*3$ 等表达式是由用户输入的，而其他内容则是由程序输出的。我们选择输出“`Expression:`”提示用户输入表达式，当然可以使用“`Please enter an expression followed by a newline`”，但显得过于冗长，没有意义；另外，短提示符“`>`”又显得过于模糊。像这样尽早给出如何使用程序的例子是很重要的，这为程序最低限度应该实现哪些功能提出了非常实际的定义。在程序的设计与分析过程中，这样的实例通常被称为用例（use case）。

当第一次碰到计算器问题时，大多数人对于程序的主要逻辑提出如下想法：

```
read_a_line
calculate      // 实际计算
write_result
```

像上面这样的描述称为伪代码（pseudo code），并不是真正的程序代码。在设计的初始阶段，当对问题的定义并不完全清晰的时候，往往采用这种伪代码形式加以描述。例如，在上面的伪代码描述中，“`calculate`”是一个函数调用吗？如果是，它的参数是什么？在这个阶段回答这些问题还为时尚早。

6.3.1 第一步尝试

在这个阶段，我们并没有准备好编写计算器程序。我们对问题还没有深入思考，不过思考总是比较困难的，而且像大多数程序员一样，我们急于编写程序代码。下面让我们试一试，编写一个简单的计算器程序，看看它将我们引向哪里。按照最初想法设计的程序如下：

```
#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter expression (we can handle + and -): ";
    int lval = 0;
    int rval;
    char op;
    int res;
    cin >> lval >> op >> rval;    // 读入表达式，如 1 + 3

    if (op == '+')
        res = lval + rval;      // 相加
    else if (op == '-')
        res = lval - rval;    // 相减
    cout << res << endl;
}
```

```

res = lval - rval; // 相减

cout << "Result: " << res << '\n';
keep_window_open();
return 0;
}

```

上面的程序读取运算符隔开的两个运算对象（如 2+2），计算并打印结果值（本例为 4），其中运算符左边的变量名为 lval，右边的变量名为 rval。

这个程序能够运行了！但如果这个程序不完整将会怎样？让程序运转起来感觉是很棒的！也许程序设计和计算机科学并不像大家所说的那么难。好吧，也许是这样的，但不要过早沉迷于这小小的成功。继续下面几项工作：

1. 进一步清理代码。
2. 加入乘法和除法（如 2*3）。
3. 加入处理多个操作符的功能（如 1+2+3）。

特别地，我们应该检查输入的内容是否符合要求（但由于匆忙，我们“忘记了”）。另外，如果一个变量的值可能是多个常量之一，检测它的值最好采用 switch 语句而不是 if 语句。

对于“1+2+3+4”这种包含多个运算符的表达式，按照它们的输入顺序进行加法运算，也就是说，从 1 开始，输入 +2 后计算 1+2（得到中间结果 3），输入 +3 后将其加到中间结果上，直到运算结束。经过尝试并修改一些简单的语法和逻辑错误之后得到如下程序：

```

#include "std_lib_facilities.h"

int main()
{
    cout << "Please enter expression (we can handle +, -, *, and /)\n";
    cout << "add an x to end expression (e.g., 1+2*3x): ";
    int lval = 0;
    int rval;
    cin >> lval; // 读入最左边的操作数
    if (!cin) error("no first operand");
    for (char op; cin >> op; ) { // 读入运算符和右操作数
        // 重复该过程
        if (op != 'x') cin >> rval;
        if (!cin) error("no second operand");
        switch(op) {
            case '+':
                lval += rval; // 相加 : lval = lval + rval
                break;
            case '-':
                lval -= rval; // 相减 : lval = lval - rval
                break;
            case '*':
                lval *= rval; // 乘 : lval = lval * rval
                break;
            case '/':
                lval /= rval; // 除 : lval = lval / rval
                break;
            default: // 没有运算符了：输出计算结果
                cout << "Result: " << lval << '\n';
                keep_window_open();
                return 0;
        }
    }
}

```

```

    error("bad expression");
}

```

程序没有错，但当我们输入 $1+2*3$ 时输出的结果是 9，而不是正确结果 7。同样，表达式 $1-2*3$ 的计算结果为 -3 而不是正确结果 -5。问题在于我们的计算顺序是错误的： $1+2*3$ 是按照 $(1+2)*3$ 的顺序计算的，而不是通常的 $1+(2*3)$ 。同样， $1-2*3$ 是按照 $(1-2)*3$ 而不是 $1-(2*3)$ 的顺序计算的。真糟糕！这种延续了几百年的人们所习惯的运算规则不会因为要简化我们的程序设计而消失，因此我们不能对“乘法的优先级高于加法”这种约定熟视无睹。

6.3.2 单词

我们必须“向前”看这一行表达式中有没有乘法或者除法运算符。如果有，必须调整这种简单的从左到右的计算顺序。然而，当我们试图这样做时，立刻遇到了很多困难。

1. 我们并没有必须要求表达式在一行输入，例如：

```

1
+
2

```

目前的代码能够正确地计算其结果。

2. 如何在数字之间搜索“*”（或者“/”）操作符，而且该表达式有可能分散在多行？
3. 如何记住“*”操作符的位置？
4. 如何不按从左到右的顺序计算表达式的值（如 $1+2*3$ ）？

让我们做一回极端的乐观主义者，首先解决前三个问题，不去担心最后一个问题，我们将在非常晚的时候考虑它。

我们四处寻求帮助，肯定有人知道如何从输入读取包括数字和操作符在内的表达式的方法，而且以一种看起来非常合理的方式进行存储。答案就是“分词”(tokenize)：读取输入字符并组合为单词(token)，因此如果键入：

45+11.5/7

程序将产生一个单词列表

```

45
+
11.5
/
7

```

 单词(token)是表示可以看作一个单元的一个字符序列，例如数字或者运算符，这也是C++编译器处理源代码的方法。实际上，“分词”在某种形式上是文本分析经常采用的方法。以C++表达式为例，可以看出所需的三种单词类型：

- 浮点常量：如C++定义的3.14、0.274e2和42。
- 运算符：+、-、*、/、%。
- 括号：(、)。

浮点常量看起来是个问题：读取12比12.3e-3容易得多，但计算器确实应该做浮点运算。同样，我们的程序必须能识别括号，否则计算器会显得没有用处。

如何在程序中表示这样的单词？试图记录每个单词的开始（和结束）会非常繁琐、杂乱，尤其是在允许表达式跨行的情况下。而且，如果将数保存为字符串，之后必须恢复它的

数值。也就是说，如果将数 42 存储为字符 4 和 2，以后必须计算这些字符所表示的数值 42，即 $4 * 10 + 2$ 。一种较好的解决方法是将每个单词表示为 (*kind*, *value*) 对，*kind* 表示单词是一个数字、运算符还是括号。对于数（在本例中，也只有数），*value* 保存的就是它的数值。

那么，如何在代码中表示一个 (*kind*, *value*) 对？我们定义一个类型 **Token** 来表示单词。为什么要这么做？记住我们为什么使用类型：它们定义了所需要的数据以及对数据能执行的有效操作。例如，**int** 类型定义了整数及其加、减、乘、除和模等运算，而 **string** 类型定义了字符串及其连接、下标等操作。C++ 语言及其标准函数库提供了很多类型，如 **char**、**int**、**double**、**string**、**vector** 和 **ostream** 等，但没有 **Token** 类型。事实上，我们希望使用的类型成千上万甚至更多，但语言及其标准函数库都未能提供，其中比较有用的是第 24 章的 **Matrix** 类型、第 9 章的 **Date** 类型以及无限精度整数类型（请尝试在互联网中搜索“**Bignum**”）等。你将会意识到一种语言不可能提供成千上万的类型：谁来定义和实现这些类型？你怎样找到这么多类型？参考手册将会多么厚？等等。C++ 等高级语言通过让用户在需要的时候自己定义类型（user-defined type）而成功解决了这个问题。

6.3.3 实现单词

在程序中的单词应该是什么样的？换句话说，自定义的 **Token** 类型是什么样的？**Token** 必须能够表示运算符（如 +、-）和数值（如 42、3.14），即表示单词是什么类别以及保存单词的数值（如果有的话）。

Token:	Token:
kind:	kind:
plus	number
value:	value:
42	3.14

在 C++ 代码中有很多方式来表示这些类型，下面是最简单实用的一种方式：

```
class Token { // 一个非常简单的用户自定义类型
public:
    char kind;
    double value;
};
```

Token 是一个类型（类似于 **int** 或者 **char**），因此可用于定义变量及值。它有两个部分（称为成员）：**kind** 和 **value**。关键字 **class** 表示定义一个“用户自定义类型”，该类型可以有成员，也可以没有成员。第一个成员 **kind** 是一个 **char** 字符，因此可以方便地保存 ‘+’ 和 ‘*’ 表示加法和乘法。我们可以如下方式进行类型定义：

```
Token t;           // t 是 Token 类型
t.kind = '+';     // t 表示 “加法”
Token t2;          // t2 是另一个 Token 类型
t2.kind = '8';    // 这里用字符 '8' 表示数值的类型
t2.value = 3.14;
```

我们使用成员访问符号 “*object_name.member_name*” 访问成员，可以将 **t.kind** 读作 “**t's kind**”，将 **t2.value** 读作 “**t2's value**”。此外，可以像复制 **int** 型对象一样复制 **Token** 对象：

```
Token tt = t;      // 拷贝初始化
if (tt.kind != t.kind) error("impossible!");
t = t2;            // 赋值
cout << t.value;   // 将会输出 3.14
```

有了 Token 类型，我们可以将表达式 $(1.5+4)*11$ 表示为如下 7 个单词：

'('8'	'+'	'8')'	'*'	'8'
1.5		4			11	

注意，“+”这种简单的单词不需要值，因此我们不使用它的 value 成员。我们需要一个字符来表示“数值”单词，由于“8”很明显不是一个运算符或者标点符号，这里选它标识“数值”单词。使用“8”来表示“数”有点含混，但暂时先这么用。

Token 是 C++ 用户自定义类型的一个实例。一个用户自定义类型可以有成员函数（操作）和数据成员，定义成员函数的理由有很多。对于 Token，我们不需要定义函数，因为已经提供了简单用户定义类型的读写成员的默认方式。

```
class Token {
public:
    char kind;      // 单词类型
    double value;  // 对数值类型的单词：记录它的值
};
```

现在我们可以初始化（“构造”）Token 对象。例如：

```
Token t1 {'+'};      // 初始化 t1 使得 t1.kind = '+'
Token t2 {'8', 11.5}; // 初始化 t2 使得 t2.kind = '8' 并且 t2.value = 11.5
```

对于构造函数的更多内容请参考 9.4.2 节和 9.7 节。

6.3.4 使用单词

现在，或许我们能够实现完整的计算器程序了。然而，可能前面的设计只有一小部分有用。在计算器程序中如何使用 Token？我们可以将输入读到一个 Token 的 vector 中：

```
Token get_token(); // 从 cin 读入单词的函数

vector<Token> tok; // 单词存储在 vector 中

int main()
{
    while (cin) {
        Token t = get_token();
        tok.push_back(t);
    }
    // ...
}
```

接下来，我们可以先读取一个表达式，然后对其进行运算。例如，对于 $11*12$ 可以得到：

'8'	'*'	'8'
11		12

我们可以从中找到乘法符号及其操作数，因此非常容易地执行乘法运算，因为数字 11 和 12 是作为数字而不是字符串存储的。

接下来看一个更复杂的表达式，如 $1+2*3$ ，则 tok 包含 5 个 Token：

'8'	'+'	'8'	'*'	'8'
1		2		3

通过一个简单的循环就可以找到乘法操作符：

```
for (int i = 0; i<tok.size(); ++i) {
    if (tok[i].kind=='*') {           // 找到一个乘法运算!
        double d = tok[i-1].value*tok[i+1].value;
        // 接下来呢?
    }
}
```

但接下来如何处理乘积 d 呢？如何确定子表达式的计算顺序呢？因为加法运算符在乘法运算符之前，所以不能直接按照从左到右的顺序计算。我们可以试试从右到左计算！但这么做对 $1+2*3$ 是正确的，对 $1*2+3$ 又是错误的了。更糟的是 $1+2*3+4$ ，必须“由内至外”计算： $1+(2*3)+4$ 。又出现了新的问题，我们如何处理括号呢，最终我们到底应该如何做呢？似乎是走进了死胡同。我们必须后退一步，先停止程序的编写，重新考虑如何读取和分析输入表达式，并计算它的值。

我们对于此问题求解（编写计算器程序）的第一次尝试以满怀热情开始，但现在已经走到尽头了。对第一次编程来说，这是很常见的。这不是灾难，它使我们加深了对这个问题的理解。而且在本例中，这次尝试还给我们带来了一个有用的概念：单词。我们今后会反复遇到 $(name, value)$ 对这种形式，单词是一个很好的实例。但是，我们必须确保这种轻率的、无计划的“编码”不会浪费太多时间。正确的做法是，在做过分析（理解问题）和设计（决定解决方案的整体结构）以后才进行程序设计。



试一试

另一方面，为什么不能找一个更简单的方法来解决这个问题？这看起来并不是那么困难。即便尝试没有什么效果，也可以增进我们对问题和最终求解方案的理解。现在马上思考你可以做些什么。比如对于表达式 $12.5+2$ ，我们可以先进行单词划分，然后确定表达式很简单，最终计算出结果。这个过程有一点凌乱，但它比较直接，或许我们可以沿着这个思路继续前进，找到很好的解决方法。接着考虑这种情况：在 $2+3*4$ 中我们既发现了‘+’又发现了‘*’，应该如何处理呢？仍然能够通过“蛮力”方式处理。但是，对于 $1+2*3/4%5+(6-7*(8))$ 这种更复杂的表达式又如何处理呢？如何处理像 $2+*3$ 和 $2&3$ 这样的错误呢？稍微花些时间思考一下，或许可以在纸上写点什么，比如勾勒一下可能的解决方案，写出一些有趣或重要的输入表达式。

6.3.5 重新开始

现在再看一下问题，不要急于得出不完善的解决方案。必须注意，如果这个程序（计算器）运行后只计算一个表达式的值就结束，显然是不满足要求的。我们希望程序的一次执行能够对若干表达式进行计算，因此改进伪代码如下：

```
while (not_finished) {
    read_a_line
    calculate      // 实际计算
    write_result
}
```

很明显程序变得复杂了，但想想我们使用计算器的情景，你就会意识到一次做多个运算是很

平常的事情。难道我们让用户做一次运算就启动一次程序吗？可以，但是在很多现代操作系统上启动程序的过程比较慢，因此最好不要这样做。

当我们审视上面的伪代码、最初的解决方案以及设计的使用实例，就产生如下几个问题（其中某些给出了初步答案）：

1. 如果输入表达式 $45+5/7$ ，那么如何找出其中的个体元素：45、+、5、/ 和 7？（单词化！）
2. 如何标识表达式的结束？当然是用换行符！（要始终保持对“当然”的怀疑：“当然”不是一个有力的理由。）
3. 如何将表达式 $45+5/7$ 作为数据存储以便于计算？在计算加法之前必须先将字符 4 和 5 转换为整数 45，即 $4*10+5$ 。（因此单词化是解决方案的一部分。）
4. 如何保证表达式 $45+5/7$ 的计算顺序为 $45+(5/7)$ 而不是 $(45+5)/7$ ？
5. 表达式 $5/7$ 的值是多少？大约是 0.71，但这不是一个整数。根据对计算器的使用经验可知，用户往往会期望得到一个浮点计算结果。我们应该允许输入表达式中出现浮点数吗？当然！
6. 可以使用变量吗？例如，我们能否使用下面的表达式：

```
v=7
m=9
v*m
```

好主意，不过先放一放，我们还是先实现计算器的基本功能。

如何回答问题 6 可能是求解方案中最重要的抉择。在 7.8 节，你会看到，如果我们决定实现变量功能，程序代码量将会是原来的两倍。让最初版本正常运行起来所花费的时间，也将是原来的两倍。以我们的估计，如果你是一个新手，将会付出四倍的工作量，因而很可能最终放弃。在程序设计早期避免“功能蔓延”是很重要的，应该保证先构建一个简单的版本，只实现最基本的功能。一旦程序能够运转，你可以有更大的野心继续完善程序。分阶段实现一个程序比一次完成要简单得多。如果一开始就实现变量功能还有一个负面影响：很难抵挡进一步添加“漂亮特性”的诱惑。加上常用的数学函数如何？再加上循环功能怎么样？一旦开始便很难停下来。

从程序员的观点来看，问题 1、3 和 4 是令人困扰的。这几个问题相互关联，因为一旦找到一个 45 或者一个 ‘+’，我们应该如何处理它们？也就是说，在程序中如何存储它们？很明显，单词化是整个解决方案的一部分，但仅仅是一部分而已。

 一个有经验的程序员如何应对这些问题？当面对一个棘手的技术问题时，通常都有一个标准答案。我们知道，从计算机能够通过键盘接收符号输入开始，人们就已经开始编写计算器程序了。至少已经有了 50 年历史，因此肯定有很成熟的解决方案。在这种情况下，有经验的程序员就会咨询同事、查阅文献。希望猛冲猛闯，一夜之间打破 50 年来的经验是很傻的想法。

6.4 文法

对于如何理解表达式的含义，已经有标准的解决方法了：首先读入符号，将它们组合为单词。因此，如果键入

45+11.5/7

程序应该产生如下单词列表：

```
45
+
11.5
/
7
```

一个单词就是一个字符序列，用来表示一个基本单元，例如数字或者运算符。

在产生单词之后，我们的程序必须保证对整个表达式正确解析。例如，我们知道表达式 $45+11.5/7$ 的计算顺序是 $45+(11.5/7)$ 而不是 $(45+11.5)/7$ ，但如何告诉程序这些有用的规则呢（例如除法比加法优先级高）？标准的方法就是设计一个文法（grammar）来定义表达式的语法，然后在程序中实现这些文法规则。例如：

```
// 简单的表达式文法：
```

```
Expression:
  Term
  Expression "+" Term      // 加
  Expression "-" Term      // 减

Term:
  Primary
  Term "*" Primary         // 乘
  Term "/" Primary         // 除
  Term "%" Primary         // 求余(模)

Primary:
  Number
  "(" Expression ")"       // 分组

Number:
  floating-point-literal
```

这是一个简单的规则集合，最后一条规则读作“一个 **Number** 是一个浮点常量”，倒数第二条规则表明“一个 **Primary** 是一个 **Number**，或者是'('后接一个 **Expression** 再接一个')'”。针对 **Expression** 与 **Term** 的规则类似，都是依赖其后的规则来定义。

如 6.3.2 节，我们从 C++ 定义中借用了如下几类单词：

- 浮点常量：与 C++ 定义相同，如 3.14、0.274e2 和 42 等。
- 运算符：+、-、*、/、% 等。
- 括号：(、)。

从最初的伪代码到现在使用单词和文法的方法，在概念上是一个巨大的飞跃。这正是我们所期望的那种飞跃，但不依靠帮助——前人的经验、参考文献和导师的指导是办不到的。

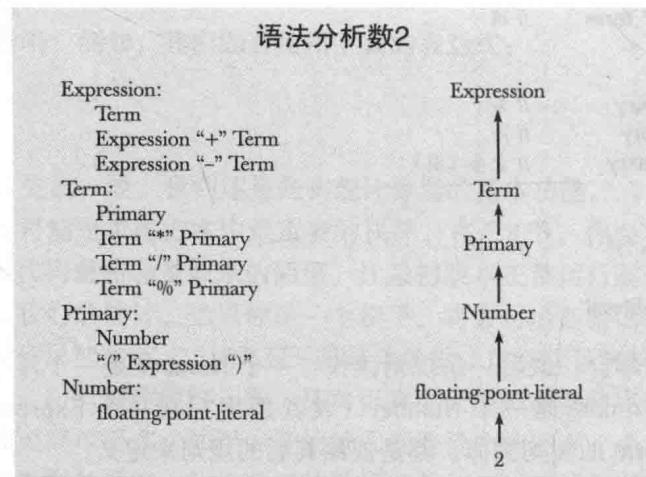
乍看起来，文法好像根本没有意义，技术符号通常都是如此。然而，请记住它是一种通用的、优美的符号（最终你会体会到这种符号的好处），实际上，使用这样一套符号系统，你在中学时期或更早就有能力做。你自己来计算 $1-2*3$ 、 $1+2-3$ 和 $3*2+4/2$ 这样的表达式，显然是没有问题的，如何计算已经深深印在你的头脑中了。但你能解释你是如何做的吗？你能给一个从未学过传统算术的人解释清楚吗？你的方法能用来计算任意运算符和运算数组合出的表达式吗？为了能清楚地表达计算方法，而且足够详细、足够精确，能被计算机所理解，我们需要一种符号系统——对此，文法是一种常规的、强有力的工具。

如何来读入一个文法呢？基本方法是这样的：对于给定输入，从顶层规则 **Expression** 开始，搜索与输入单词匹配的规则。根据文法读取单词流的方式称为语法分析（parsing），实

现该功能的程序称为分析器 (parser) 或者语法分析器 (syntax analyzer)。语法分析器从左到右读取单词，与我们输入和阅读的顺序一致。下面给出一个非常简单的实例：2是一个表达式吗？

1. 一个 Expression 必须是一个 Term 或者以 Term 结尾，一个 Term 必须是一个 Primary 或者以 Primary 结尾，一个 Primary 必须以 ‘(’ 或者 Number 开头。很明显，2 虽然不是 ‘(’，但它是一个浮点常量 (floating-point-literal) 型 Number，因此它是一个 Primary。
2. Primary (Number 2) 前没有 /、* 或者 %，因此它是一个完整的 Term，而不是 /、* 或者 % 表达式的结尾。
3. Term (Primary 2) 前没有 + 或者 -，因此它是一个完整的 Expression，而不是 + 或者 - 表达式的结尾。

因此，根据文法，2 是一个表达式，分析步骤如下所示：

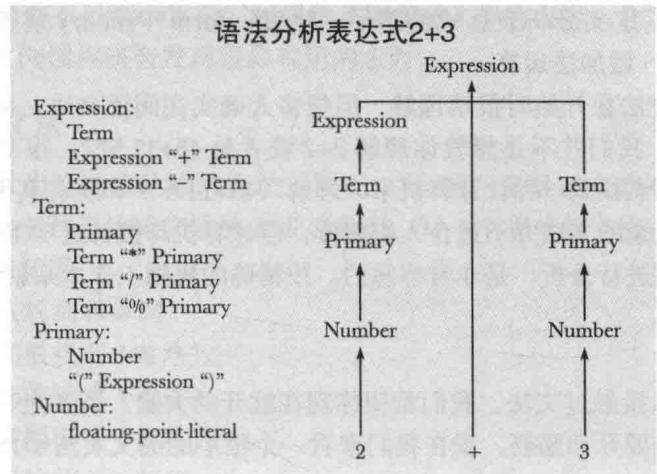


上面给出了根据定义解析表达式的方法，由解析路径可知 2 是一个表达式，因为 2 是一个浮点常量，而一个浮点常量是一个 Number，一个 Number 是一个 Primary，一个 Primary 是一个 Term，一个 Term 是一个 Expression。

下面给出一个更复杂的实例：2+3 是一个 Expression 吗？很明显，许多推导过程与分析 2 时一样：

1. 一个 Expression 必须是 Term 或者以 Term 结尾，Term 必须是 Primary 或者以 Primary 结尾，Primary 必须以 ‘(’ 开头或者是 Number。显然 2 不是左括号，而是一个浮点常量类型的 Number，因而是一个 Primary。
2. Primary (Number 2) 前面没有 /、* 或者 %，因此它是一个完整的 Term，而不是 /、* 或者 % 表达式的结尾。
3. Term (Primary 2) 后面跟着 + 运算符，因此它是表达式第一部分的结尾，必须寻找 + 运算符后面的 Term。与推导 2 是一个 Term 的方式一样，3 也是一个 Term。由于 3 后面没有 + 或 - 操作符，因此它是一个完整的 Term，而不是加 / 减表达式的一部分。因此，2+3 符合 Expression+Term 规则，因此是一个 Expression。

我们还是以图形方式说明这个推导过程，为简化省略了浮点常量到 Number 规则的推导。

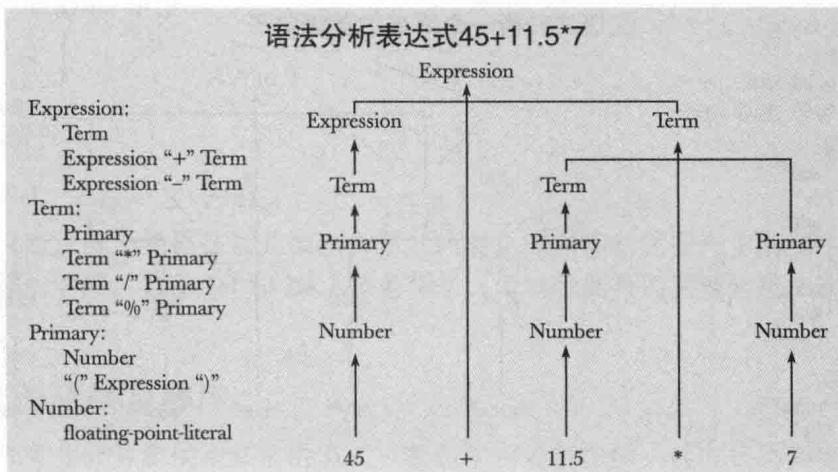


上图给出了根据定义分析表达式的路径，由分析路径可知 $2+3$ 是一个**Expression**，因为2是一个**Term**类型的表达式，3是一个**Term**，而**Expression**后接一个‘+’再接一个**Term**构成一个**Expression**。

我们对文法感兴趣的真正原因在于它可以帮助我们正确地分析同时包含+和*的表达式，下面看看如何处理 $45+11.5*7$ 。然而，计算机利用上述规则分析此表达式的详细过程是令人乏味的，因此我们省略其中一些熟悉的中间步骤，如分析2和 $2+3$ 的过程。很明显，45、11.5和7都是浮点常量，因而都是**Number**，也都是**Primary**，因此我们忽略了所有**Primary**之下的规则。于是有：

1. 45是一个**Expression**，后面紧跟一个+，因此需要寻找一个**Term**，以便实现**Expression+Term**文法规则。
2. 11.5是一个**Term**，后面紧跟一个*，因此需要找到一个**Primary**，匹配**Term*Primary**文法规则。
3. 7是一个**Primary**，由**Term*Primary**规则可知 $11.5*7$ 是一个**Term**。同样，根据**Expression+Term**规则可知， $45+11.5*7$ 是一个**Expression**。特别地，这个表达式需要先算 $11.5*7$ ，然后再运算 $45+11.5*7$ ，正像我们所写的 $45+(11.5*7)$ 。

下面给出推导过程的图示（省略了浮点常量到**Number**规则的推导）：



上面给出了根据定义分析表达式的路径。注意：`Term*Primary` 规则表明了 11.5 先与 7 做乘法，而不是与 45 做加法运算。

你会发现这种方法在开始时很难理解，但很多人确实在阅读文法，而简单的文法理解起来并不困难。然而，我们并不是想教你理解 $2+2$ 或者是 $45+11.5*7$ 。很明显，你已经知道这些了。我们只是在尝试找到一种让计算机来“理解” $45+11.5*7$ 和所有其他更加复杂的表达式的方法。实际上，复杂的文法并不适合人们阅读，但计算机却擅长这项工作。让计算机快速、准确地遵循这些规则进行分析，是非常容易的。按精确的规则工作本来就是计算机所擅长的。

6.4.1 英文文法

如果你以前从未接触过文法，我们希望你现在就开动大脑。事实上，即使你以前曾经接触过文法，现在还是要开动脑筋，现在我们来看一个很小的英文文法子集：

Sentence:

Noun Verb // 例如, C++ rules

Sentence Conjunction Sentence // 例如, Birds fly but fish swim

Conjunction:

"and"

"or"

"but"

Noun:

"birds"

"fish"

"C++"

Verb:

"rules"

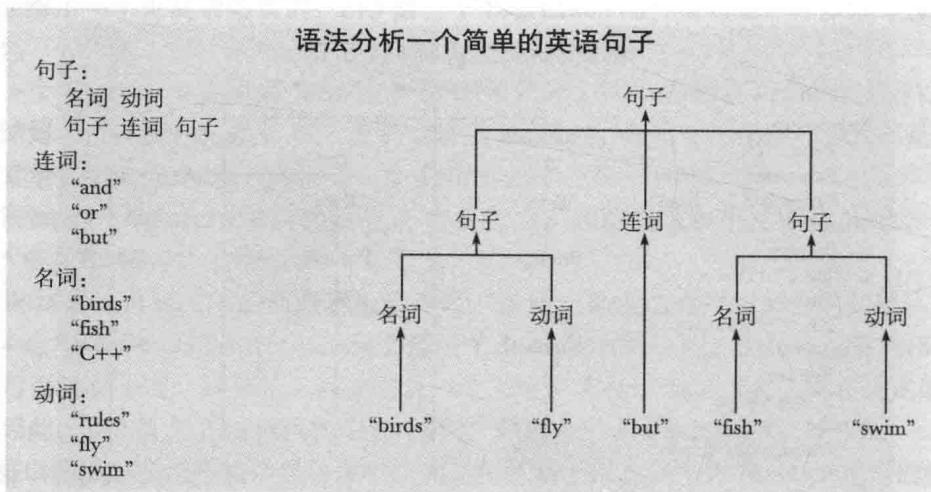
"fly"

"swim"

一个句子由语言的基本单元（例如名词、动词与连接词）构成。根据语法规则可以分析一个句子，确定哪些是名词，哪些是动词，等等。这个简单的文法也会产生一些没有意义的句子，例如：“C++ fly and birds rules”，但如何修正这一问题已不属于本书的范围。

许多人已经在中学或者外语课（例如英语课程）上学习过这些文法规则了，这些文法规则是非常基本的。实际上，这些规则深深地印在我们的大脑之中，这也是神经学所研究的重要课题。

下面来看一下语法分析树，前面我们用它来分析表达式，这里用来描述简单的英语：



这些语法看起来并不是那么复杂。如果你不理解 6.4 节的内容，你现在可以回去头去重新阅读，经过第二次阅读你将会发现很多有用的东西。

6.4.2 设计一个文法

我们是如何设计出这些表达式的文法规则呢？“经验”是最诚实的回答，我们的方法不过就是人们所习惯的书写表达式文法的方法。然而，写一个简单的文法还是有一些相当直接的方法的，我们需要知道：

1. 如何区分文法规则和单词。
2. 如何来排列文法规则（顺序）。
3. 如何表达可选的模式（多选）。
4. 如何表达重复的模式（重复）。
5. 如何辨识出起始的文法规则。

不同的教材与不同的分析程序使用不同的符号约定和术语。例如，一些人习惯称单词为终结符（terminal），称规则为非终结符（non-terminal）或者产生式（production）。我们简单地将单词放在（双）引号中，与规则相区分。而文法第一条规则为默认的起始规则。一个规则的可选模式放在不同行中。例如：

```
List:  
  {" Sequence "}  
Sequence:  
  Element  
  Element "," Sequence  
Element:  
  "A"  
  "B"
```

因此，上面文法的含义是，一个 Sequence 是一个 Element，或者是一个 Element 后面紧跟一个 Sequence，并且两者以逗号隔开。一个 Element 是字母 A 或是字母 B。List 是在花括号中的一个 Sequence。我们可以生成一些 List（如何生成的？）：

```
{A}  
{B}  
{A,B}  
{A,A,A,A,B}
```

但下面这些不是 List（为什么？）：

```
{ }  
A  
{A,A,A,A,B  
{A,A,C,A,B }  
{A B C }  
{A,A,A,A,B, }
```

上面的文法规则不是你在幼儿园中所学过的或是深深印在你脑海中的那些，但它们也并非什么高深的学问。参见 7.4 和 7.8.1 节的例子，可以看到我们是如何用文法表述语法思想的。

6.5 将文法转换为程序

现在有许多令计算机使用文法的方式。我们采用最简单的一种方式：为每个文法规则

写一个函数，并且使用自定义类型 `Token` 表示单词。实现一个文法的程序通常被称为分析器（parser）。

6.5.1 实现文法规则

我们在计算器程序的设计中使用了四个函数，一个函数用于读入单词，其他三个函数分别实现文法的三条规则：

```
get_token()      // 利用 cin 读入字符并合并为单词
expression()    // 调用 term() 和 get_token() 来处理加和减
term()          // 调用 primary() 和 get_token() 来处理乘、除和求余
primary()       // 调用 expression() 和 get_token() 来处理数值和括号
```

注意：每个函数只处理表达式的一个特定部分，将其他工作留给其他函数，这样可以简化函数的实现。如同每个人处理自己所擅长的问题，将其他不熟悉的问题留给同事完成一样。

这些函数应该具体做什么呢？每个函数应该根据其所实现的文法规则，调用其他文法函数，并利用 `get_token()` 获得规则所需要的单词。例如，当 `primary()` 函数实现 “(Expression)” 规则时，它必须调用

```
get_token()      // 处理括号
expression()    // 处理表达式
```

这些语法分析函数的返回值又应该是什么呢？这个问题实际等价于——我们想得到什么结果呢？例如，对于 `2+3`，`expression()` 应该返回 `5`。毕竟，所有信息都包含其中了。这就是我们应该做的！这样做实际上回答了前面列表中最复杂的问题：“如何表示 `45+5/7` 才有利于计算它的值？”我们在读入表达式时就计算它的值，而不是把它的某种表示形式存储到内存中。这个小小的想法其实是一个重要的突破！我们如果让 `expression()` 返回某种复杂的形式，随后再进行计算的话，程序规模会是直接计算值的版本的 4 倍。直接计算表达式的值会节省我们 80% 左右的工作量。

剩下的那个函数是 `get_token()`：由于它只处理单词，而不是表达式，因此它不能返回一个子表达式的值。例如，“`+`”和“`(`”不是表达式。因此，它必须返回一个 `Token` 对象。因此有：

```
// 匹配文法规则的函数：
Token get_token()      // 读入字符并合并为单词
double expression()    // 处理加和减
double term()          // 处理乘、除和求余
double primary()       // 处理数值和括号
```

6.5.2 表达式

下面首先编写 `expression()`，它的文法规则如下：

```
Expression:
Term
Expression '+' Term
Expression '-' Term
```

由于这是我们第一次尝试将一组文法规则转换为代码，将会经历一些不成功的开始。这是学习一种新技术常见的过程，我们可以从中学到很多有用的东西。特别地，通过观察相似

代码段表现出令人吃惊的不同行为，初学者可以学到很多。阅读代码是积累编程技巧的有效途径。

6.5.2.1 表达式：第一次尝试

首先看一下 Expression '+' Term 规则，我们首先调用 expression()，然后寻找 +(和 -)，最后是调用 term()：

```
double expression()
{
    double left = expression(); // 读入并计算一个表达式
    Token t = get_token(); // 获取下一个单词
    switch (t.kind) {
        // 查看是哪个单词
        case '+':
            return left + term(); // 读入并计算一个项，然后相加
        case '-':
            return left - term(); // 读入并计算一个项，然后相减
        default:
            return left; // 返回该表达式的值
    }
}
```

这个程序看起来不错。它几乎是文法的一个简单誊写，其结构确实非常简单：首先读入一个 Expression，然后判断它后面是否跟着一个 ‘+’ 或者一个 ‘-’，如果确是这样，则再读取 Term。

不幸的是，这里存在很大问题。怎样才能知道一个表达式的结尾在何处，以便于寻找一个 ‘+’ 或者一个 ‘-’ 呢？请记住：我们的程序从左到右读取输入，它不能预取符号来查看前面是否有 ‘+’ 运算符。事实上，这个 expression() 函数只能执行到第一行代码，因为 expression() 在一直不停地调用自己，这种情况称为无限递归（infinite recursion）。实际上递归调用还是会停止的，因为每次调用都会消耗一定内存空间，当计算机内存被耗光时，程序就会退出。“递归”的含义就是程序调用自身，并不是所有的递归都是无限的，递归是一种非常有用的程序设计技术（参见 8.5.8 节）。

6.5.2.2 表达式：第二次尝试

既然如此，我们能做什么呢？每一个 Term 都是一个 Expression，但一个 Expression 未必是 Term。也就是说，我们可以从寻找一个 Term 开始，但只有在找到一个 ‘+’ 或者一个 ‘-’ 时才寻找一个完整的 Expression。例如：

```
double expression()
{
    double left = term(); // 读入并计算一个项
    Token t = get_token(); // 获取下一个单词
    switch (t.kind) {
        // 查看是哪个单词
        case '+':
            return left + expression(); // 读入并计算一个表达式，然后相加
        case '-':
            return left - expression(); // 读入并计算一个表达式，然后相减
        default:
            return left; // 返回该项的值
    }
}
```

实际上，这个函数或多或少是可以正确运行的。我们在最终的程序中测试过它，它完全可以分析我们输入的每一个合法的表达式，甚至还能正确计算大多数表达式的值。例如：

对于 $1+2$, 首先读入一个 **Term** (值为 1), 接着是一个 ‘+’ 运算符, 然后是一个 **Expression** (恰好是一个值为 2 的 **Term**), 最后计算出结果 3。同样, $1+2+3$ 的计算结果为 6。我们还可以举出很多它可以正确运算的例子, 但我们还是简洁些: 对于 $1-2-3$, 会得到什么结果呢? 这个 **expression()** 函数会把 1 作为一个 **Term** 读入, 接着读入表达式 $2-3$ (由 **Term** 2 和后面 **Expression** 3 构成), 然后用 1 减去 $2-3$ 的值。换句话说, 它计算的是 $1-(2-3)$ 的值, 其计算结果为 2 (正数 2)。但是, 我们在小学甚至更早就学过 $1-2-3$ 等价于 $(1-2)-3$, 其结果是 -4 (负数 4)。

因此, 我们得到的是一个看似正确却不能得到正确结果的程序, 这是很危险的。这个例子尤其危险的地方在于, 它能够在很多情况下给出正确的结果。例如: 它能计算出 $1+2+3$ 正确的结果 6, 因为 $1+(2+3)$ 等价于 $(1+2)+3$ 。重要的是, 从程序设计的角度来看, 我们做错了什么吗? 每当发现错误时, 我们都应该问自己这个问题。这样可以帮助我们避免一而再、再而三地犯同样的错误。

最基本的做法是阅读代码并猜测错误在哪里, 但通常这不是一个好方法。因为我们必须理解程序代码在做什么, 必须能解释它为什么对有些表达式计算正确, 对有些表达式则计算错误。

错误分析通常也是能找出正确求解方案的最好方法。在本例中, 我们定义 **expression()** 函数如下: 先读入一个 **Term**, 接着判断它后面是否有一个 ‘+’ 或者一个 ‘-’, 若有则寻找一个 **Expression**。实际上这实现了一个略微不同的文法:

```
Expression:
  Term
  Term '+' Expression      // 加
  Term '-' Expression      // 减
```

这与我们希望实现的语法的不同之处在于, 对 $1-2-3$ 这样的表达式, 我们希望解释为 **Expression** 1-2 后接 ‘-’ 再接 **Term** 3, 但得到的却是 **Term** 1 后接 ‘-’ 再接 **Expression** 2-3; 也就是说, 我们希望 $1-2-3$ 意味着 $(1-2)-3$, 但得到的却是 $1-(2-3)$ 。

是的, 调试可能是一项非常乏味、棘手的工作, 也非常耗时。但在此例中, 我们确实在使用小学就学过的、可以帮我们避免很多错误的运算规则。潜在的困难——可能出错的地方在于我们必须把这些规则教给计算机, 但计算机却不善于学习这些内容。

注意到, 我们实际上可以将 $1-2-3$ 定义为 $1-(2-3)$ 而不是 $(1-2)-3$, 那我们就不再讨论这个问题了。但这显然是不行的, 我们不能改变人们习惯的算术运算规则。这就是困难所在: 常见的程序设计难题, 多数是因为程序必须符合传统的规则, 而这些规则早在计算机出现之前就已经建立起来并被人们所使用了。

6.5.2.3 表达式: 幸运的第三次

那么现在应该做什么呢? 再次回顾一下文法 (6.5.2 节中那个正确文法): 任何一个 **Expression** 都以一个 **Term** 开始, **Term** 后面可以跟一个 ‘+’ 或者一个 ‘-’。因此, 我们必须先寻找 **Term**, 看它后面是否有一个 ‘+’ 或者一个 ‘-’, 并且重复此步骤直到 **Term** 后面没有加号或者减号为止。例如:

```
double expression()
{
    double left = term();           // 读入并计算一个项
    Token t = get_token();          // 获取下一个单词
```

```

while (t.kind=='+' || t.kind=='-') { // 寻找 + 或 -
    if (t.kind == '+')
        left += term(); // 计算一项的值并相加
    else
        left -= term(); // 计算一项的值并相减
    t = get_token();
}
return left; // 最终：没有更多 + 或 -；返回结果
}

```

这个程序可能有点混乱：我们必须引入一个循环来寻找加号与减号。而且，这里还有一些重复工作：对‘+’和‘-’的测试进行了两次，`get_token()` 函数也被调用了两次。这样导致程序的逻辑变得有点混乱，下面我们修改程序，去掉对‘+’和‘-’的重复测试。

```

double expression()
{
    double left = term(); // 读入并计算一个项
    Token t = get_token(); // 获取下一个单词
    while (true) {
        switch (t.kind) {
            case '+':
                left += term(); // 计算一项的值并相加
                t = get_token();
                break;
            case '-':
                left -= term(); // 计算一项的值并相减
                t = get_token();
                break;
            default:
                return left; // 最终：没有更多 + 或 -；返回结果
        }
    }
}

```

注意：除了循环，该程序与第一次尝试的程序非常相似（见 6.5.2.1 节）。我们所做的就是用循环语句替代了 `expression()` 中对自身的调用。换句话说，我们把 Expression 文法规则中的 Expression 转换为循环语句，在循环语句中寻找后接‘+’和‘-’的 Term。

6.5.3 项

Term 的文法规则与 Expression 规则非常相似：

Term:

```

Primary
Term '*' Primary
Term '/' Primary
Term '%' Primary

```

因此，它们的代码基本相同。下面是第一次尝试：

```

double term()
{
    double left = primary();
    Token t = get_token();
    while (true) {
        switch (t.kind) {

```

```

    case '!':
        left *= primary();
        t = get_token();
        break;
    case '/':
        left /= primary();
        t = get_token();
        break;
    case '%':
        left %= primary();
        t = get_token();
        break;
    default:
        return left;
}
}

```

不幸的是，程序没有编译成功：编译器给出了错误信息——C++ 对浮点数没有定义模运算（%）。当我们回答 6.3.5 节列出的第五个问题时“我们应该允许输入表达式中出现浮点数吗？”，我们做出了肯定的回答“当然！”，实际上我们当时并没有全面考虑这个问题，从而陷入了功能蔓延的困境。这种情况经常会发生！那么我们应该如何处理呢？我们可以在运行时检查运算符 % 的两个运算数是否为整数，若不是则给出错误信息；或者简单地将操作符 % 排除在外，本书中就选择这种简单方法。我们可以随时将运算符 % 加进来（参见 7.5 节）。

在去掉运算符 % 以后，函数能够正常运行了，能够正确分析 Term 并进行计算。然而，有经验的程序员会注意到 term() 中存在一个不可接受的情况。如果我们输入 2/0 会发生什么情况？C++ 程序中零不能作为除数，否则计算机硬件会检测出这一情况，并终止程序，给出一些无用的错误信息。一个新手很难发现问题在哪里，所以，最好在程序中检查这种情况并给出一个恰当的错误提示。

```

double term()
{
    double left = primary();
    Token t = get_token();
    while (true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = get_token();
                break;
            case '/':
                {   double d = primary();
                    if (d == 0) error("divide by zero");
                    left /= d;
                    t = get_token();
                    break;
                }
            default:
                return left;
        }
    }
}

```

为什么我们把处理 ‘/’ 的语句放入一个语句块内呢？这是编译器规定的，如果要在 switch 语句中定义和初始化变量，必须把它们放在一个语句块内。

6.5.4 基本表达式

基本表达式的文法规则也很简单：

Primary:

Number
'(' Expression ')'

它的实现代码有点混乱，因为其中有很多可能导致语法错误的地方。

```
double primary()
{
    Token t = get_token();
    switch (t.kind) {
        case '(': // 处理 '(' expression ')'
            { double d = expression();
              t = get_token();
              if (t.kind != ')') error("')' expected");
              return d;
            }
        case '8':
            return t.value; // 返回该数的值
        default:
            error("primary expected");
    }
}
```

基本上，与 expression() 和 term() 函数相比并没有什么新内容。我们使用了相同的语言指令、相同的单词处理方式以及相同的编程技巧。

6.6 试验第一个版本

这些计算器函数，需要实现 get_token() 函数并提供一个 main() 函数。main() 函数比较简单，仅仅用于 expression() 函数的调用和结果输出。

```
int main()
try {
    while (cin)
        cout << expression() << '\n';
    keep_window_open();
}
catch (exception& e) {
    cerr << e.what() << '\n';
    keep_window_open();
    return 1;
}
catch (...) {
    cerr << "exception\n";
    keep_window_open();
    return 2;
}
```

错误处理部分还是老样式（参见 5.6.3 节）。我们把 get_token() 函数的实现留到 6.8 节介绍，这里只是用它来测试计算器程序的第一个版本。



试一试

计算器程序的第一个版本（包括 `get_token()`）在文件 `calculator00.cpp` 中。请尝试编译、运行它，验证结果。

不出所料，计算器程序的第一个版本并没有很好地按我们期望的方式来工作。于是我们不禁要问“它为什么不按我们期望的方式工作呢？”或者更进一步，“它为什么像这样工作呢？”以及“它能做什么呢？”让我们输入数字 2 并换行，程序没有反应。再敲一个换行来看看程序是否进入睡眠状态了，仍然没有反应。接着输入数字 3 并换行，程序还是没有反应！再输入数字 4 接着换行，程序终于给出一个应答 2！此时屏幕显示如下：

```
2
3
4
2
```

接着继续输入 `5+6`，程序输出 5，此时屏幕显示如下：

```
2
3
4
2
5+6
5
```

除非你以前有过编程经验，否则你多半会陷入深深的迷惑之中！事实上，即使是一个有经验的程序员对此也可能感到迷惑。接下来该如何处理呢？这时你应该尝试结束程序。但应该如何结束程序呢？我们没有在程序中设置结束命令，但一个错误可以导致程序结束运行。因此，你可以输入一个 `x`，程序会输出 `Bad token` 然后结束运行。噢，终于有这么一次，程序能按我们的设想工作了！

但是，我们忘记将屏幕上的输入与输出信息加以区分了。在解决主要问题之前，让我们先对输出做些改动，以利于呈现程序做了什么事情。我们在输出内容前增加一个‘=’，将其与输入信息区分开来：

```
while (cin) cout << "=" << expression() << '\n'; // 版本 1
```

现在，重新输入与上一次运行完全一样的符号，我们得到如下结果：

```
2
3
4
= 2
5+6
= 5
x
Bad token
```

很奇怪！让我们试着理解程序做了些什么。我们还尝试了另外几个例子，但还是集中精力看看这个例子，下面几点令人迷惑不解：

第一次输入 2、3 并换行以后，为什么程序没有反应呢？

在输入 4 以后，为什么程序输出的是 2 而不是 4 呢？

在输入 5+6 以后，为什么程序回答的是 5 而不是 11 呢？

产生这些奇怪的结果有很多可能的原因，其中一些将在下一章详细讨论，这里我们只是简单思考。程序会产生算术运算错误吗？这几乎是不可能的。但确实结果是错误的：4 的值不应该是 2，5+6 的结果是 11 而不应该是 5。我们再试着输入 1 2 3 4+5 6+7 8+9 10 11 12 然后换行，看看会出现什么结果。我们将得到：

```
1 2 3 4+5 6+7 8+9 10 11 12
= 1
= 4
= 6
= 8
= 10
```

噢！没有输出 2 或者 3，而且为什么输出 4 而不是 9 (4+5) 呢？为什么输出 6 而不是 13 (6+7) 呢？仔细观察：程序在每三个单词中输出一个！是不是程序“吃掉”了一些输入而没有让它们参加运算呢？确实是这样，考虑 expression() 函数：

```
double expression()
{
    double left = term();           // 读入并计算一个项
    Token t = get_token();          // 获取下一个单词
    while (true) {
        switch (t.kind) {
            case '+':
                left += term();   // 计算一项的值并相加
                t = get_token();
                break;
            case '-':
                left -= term();   // 计算一项的值并相减
                t = get_token();
                break;
            default:
                return left;      // 最终：没有更多 + 或 -；返回结果
        }
    }
}
```

当 get_token() 返回的单词不是 ‘+’ 或者 ‘-’ 时，我们简单地从 expression() 函数返回了。我们没有使用那个单词，也没有把它保存下来用于后面的计算。这是不明智的做法，甚至没有判断单词是什么就把它丢弃的做法不是一个好的策略。快速查看一下可以发现，term() 函数中也存在同样的问题。这就解释了我们的计算器为什么会每处理一个单词后就会“吃掉”后面的两个。

我们来修改 expression() 函数，使其不再“吃掉”单词。那么当程序不需要下一个单词 (t) 时，应该把它放在哪儿呢？我们可以给出很多复杂精巧的方案，但在此选择最显而易见的一种（你一看到这个方法就会知道它确实“显而易见”）：如果其他某个函数需要该单词，而此函数从输入流读入单词的话，我们将该单词放回输入流，它就能够再次被此函数读取！实际上，我们是可以把字符退回标准输入流 istream 中的，但那不是我们真正想要的。我们希望的是处理输入的单词，而不是把输入流变得混乱。我们需要的是一个专门用于单词处理的输入流，能够将已经读出的单词重新放回去。

假设我们已经有一个称为 ts 的单词流 (Token_stream)，并假设 Token_stream 的成员函数 get() 用于返回下一个单词，成员函数 putback(t) 用于将单词 t 放回单词流。一旦了解了如

何使用单词流之后，我们将在第 6.8 节实现 Token_stream。给定了 Token_stream，我们可以重写 expression() 函数，令其将不使用的单词放回 Token_stream。

```
double expression()
{
    double left = term();           // 读入并计算一个项
    Token t = ts.get();             // 从单词流中获取下一个单词

    while (true) {
        switch (t.kind) {
            case '+':
                left += term();     // 计算一项的值并相加
                t = ts.get();
                break;
            case '-':
                left -= term();     // 计算一项的值并相减
                t = ts.get();
                break;
            default:
                ts.putback(t);      // 将 t 放回到单词流中
                return left;         // 最终：没有更多 + 或 -；返回结果
        }
    }
}
```

另外，必须对 term() 函数做同样的修改：

```
double term()
{
    double left = primary();        // 读入并计算一个项
    Token t = ts.get();             // 从单词流中获取下一个单词

    while (true) {
        switch (t.kind) {
            case '*':
                left *= primary();
                t = ts.get();
                break;
            case '/':
                { double d = primary();
                  if (d == 0) error("divide by zero");
                  left /= d;
                  t = ts.get();
                  break;
                }
            default:
                ts.putback(t);      // 将 t 放回到单词流中
                return left;
        }
    }
}
```

对最后一个分析函数 primary()，只需要把 get_token() 函数改为 ts.get()，primary() 函数使用它读入的每一个单词。

6.7 试验第二个版本

现在，我们准备测试程序的第二个版本。计算器程序（包含 Token_stream）的第二个版

本可在 calculator 01.cpp 获得。运行并试验输入 2 并换行以后，程序没有输出，再次换行程序仍然没有输出。输入 3 并换行以后程序输出 2，输入 2+2 并换行以后程序输出结果 3。此时屏幕上显示：

```
2
3
=2
2+2
=3
```

由此可知，也许在 expression() 和 term() 中使用 putback() 并没有解决问题。下面做另一个测试：

```
2 3 4 2+3 2*3
=2
=3
=4
=5
```

程序给出了正确的结果！但最后一个结果（6）却不见了。这里仍然存在一个单词预读方面的问题。但是，这次的问题不是我们的程序“吃掉”了字符，而是它不能返回表达式的运算结果，除非再输入后续表达式。也就是说，一个表达式的结果不是被立即输出，而被推迟到程序读入下一个表达式的第一单词以后才输出。不幸的是，只有在输入下一个表达式并回车以后，程序才能读到那个单词。程序本身没有错误，只是它的输出有些延迟。

如何改进这个问题？一个很明显的方法是加入一个“输出命令”。我们使用分号标识一个表达式的结束并触发结果的输出。另外，我们再增加一个“退出命令”，实现程序的正常退出。字符 q（表示“quit” – 退出）用于表示退出命令是很恰当的。在原来版本的 main() 函数中，有：

```
while (cin) cout << "=" << expression() << '\n'; // 版本 1
```

我们把它改成下面这样，可能有点复杂，但却更加实用：

```
double val = 0;
while (cin) {
    Token t = ts.get();

    if (t.kind == 'q') break;      // 'q' 表示“退出”
    if (t.kind == ';')            // ';' 表示“立即输出结果”
        cout << "=" << val << '\n';
    else
        ts.putback(t);
    val = expression();
}
```

现在的计算器程序真正可用。例如：

```
2;
=2
2+3;
=5
3+4*5;
=23
q
```

现在，我们有了一个比较好的计算器程序的初步版本。虽然还不是我们最终想要的那

样，但是可以把它作为进一步完善的基础。重要的是，现在的版本可以正常运行，然后就可以逐步改进问题、增加功能，并在这个过程中一直保持一个能正常运行的版本。

6.8 单词流

在改进计算器程序之前，我们先给出 `Token_stream` 的实现。毕竟，程序在没有获得正确输入之前是不能正确运行的。因此，我们首先实现 `Token_stream`，但并不是想偏离计算器程序这个主题太远，只是首先完成一个尽量小的可用程序。

计算器程序的输入是一个单词序列，如前面的例子 `(1.5+4)*11` 中所示（6.3.3节）。我们需要从标准输入 `cin` 中读入字符，并且能够向程序提供运行时需要的下一个单词。另外，我们发现计算器程序经常多次读入一个单词，因此应该把它们保存起来便于后续使用。这是最典型也是最基本的功能，当严格从左到右读入 `1.5+4` 时，在没有读入 ‘+’ 之前，你如何判断浮点数 `1.5` 已经完整读入了呢？实际上，在遇到 ‘+’ 之前，我们完全有可能是在读入 `1.55555`，而不是 `1.5` 的过程中。因此，我们需要一个“流”，当我们需要一个单词时，可以调用 `get()` 函数从流中产生一个单词，并且可以利用 `putback()` 把单词放回流中。根据 C++ 的语法规则，我们必须先定义 `Token_stream` 类型。

你可能注意到了前面 `Token` 定义中的 `public:`，那里使用 `public` 并没有特别的原因。但对于 `Token_stream`，则必须使用 `public` 来限定相应的函数。C++ 用户自定义类型通常由两部分构成：公有接口（用 “`public:`” 标识）和具体实现（用 “`private:`” 标识）。这样做主要为了将用户接口（用户方便使用类型所需的）和具体实现（实现类型所需的）分开，希望没有对用户的理解造成困难：

```
class Token_stream {
public:
    // 用户接口
private:
    // 实现细节
    // (对 Token_stream 的使用者来说不能直接访问)
};
```

 显然，我们经常既扮演用户的角色，又扮演实现者的角色。但是，弄清楚用户使用的公有接口和仅由实现者使用的具体实现之间的区别，对于组织程序代码是非常重要的。公有接口应该只包含用户需要的内容，经常是一组函数。私有实现包括实现公有函数所必须的内容，包括用于处理复杂细节的数据和函数，而这些都是用户不必知道也不应该直接使用的。

下面详细给出 `Token_stream` 类型。用户需要 `Token_stream` 完成什么功能？很明显，需要 `get()` 和 `putback()` 两个函数，这也是我们设计单词流这个概念的原因。`Token_stream` 能够从标准输入读入字符，从中解析出单词，因此，类中应包含能创建 `Token_stream` 对象，并令它能从 `cin` 读入字符的函数。于是，最简单的 `Token_stream` 定义如下所示：

```
class Token_stream {
public:
    Token_stream();           // 创建一个单词流，它从 cin 读入数据
    Token get();              // 获取一个单词
    void putback(Token t);   // 放回一个单词
private:
    // implementation details
};
```

这就是一个用户使用 `Token_stream` 所需要的全部内容。有经验的程序员可能会对 `cin` 是字符的唯一输入源感到惊讶，但这里我们决定只从键盘输入字符，第 7 章的一个习题将重新审视这个决定。

为什么我们使用了较长的名字 `putback()` 而不是 `put()` 呢？逻辑上看 `put()` 已经足够了。我们这样做是为了重点强调一下 `get()` 和 `putback()` 之间的不对称性，这是一个输入流，不包含能用来输出的函数。在 `istream` 中也实现了 `putback()` 函数：在系统中保持命名的一致性是比较重要的，有助于记忆和避免不必要的错误。

我们现在可以创建、使用 `Token_stream` 对象了：

```
Token_stream ts; // 一个名为 ts 的 Token_stream 对象
Token t = ts.get(); // 从 ts 获取一个单词
...
ts.putback(t); // 将单词 t 放回到 ts 中
```

下面我们要做的就是实现计算器程序的剩余部分了。

6.8.1 实现 `Token_stream`

现在，我们实现 `Token_stream` 中的三个函数。如何表示一个 `Token_stream` 呢？也就是说，需要在 `Token_stream` 中存储什么数据才能完成相应的功能呢？放回 `Token_stream` 的任何单词都需要存储空间。但为了简单起见，这里规定每次只能放回一个单词，对我们的计算器程序（和其他许多类似的程序）来说这已经够用了。因此，我们只需要声明一个单词所需的存储空间和指示该存储空间是否被占用的值。

```
class Token_stream {
public:
    Token get(); // 获取一个单词 (get() 定义见 6.8.2 节)
    void putback(Token t); // 放回一个单词
private:
    bool full {false}; // 缓冲区里是否有单词？
    Token buffer; // 这是我们存储通过 putback() 放回的单词的缓冲区
};
```

现在，我们可以来定义（“编写”）两个成员函数了，首先定义比较简单的 `putback()` 函数。`putback()` 成员函数的功能是将其参数放回 `Token_stream` 的缓冲区中：

```
void Token_stream::putback(Token t)
{
    buffer = t; // 拷贝 t 到缓冲区
    full = true; // 现在缓冲区被占用
}
```

关键字 `void` 指出 `putback()` 函数不返回任何值。

当我们在类外定义一个成员时，必须指明这个成员属于哪个类，为此，需采用如下语法：

```
class_name::member_name
```

在前面的代码中，我们以这种方式定义了 `Token_stream` 的成员函数 `putback`。

为什么我们要在类的外部定义一个成员呢？主要是为了保持代码清晰：类的定义主要说明类能够做什么。成员函数定义则指明如何做，因此，我们倾向于将其放在“别的地方”，避免和类定义混在一起分散注意。理想情况是，程序中的每个逻辑实体都很简短，能在屏幕上的一页内完整显示。如果将成员函数定义放在别处，是能做到这点的，但如果将其放在类

的定义中（“类内”成员函数定义），将很难满足这个要求。

如果我们想确认不发生这种情况——连续两次调用 `putback()` 函数且期间没有（用 `get()` 函数）读取放回流的内容，可以增加一个测试：

```
void Token_stream::putback(Token t)
{
    if (full) error("putback() into a full buffer");
    buffer = t;           // 拷贝 t 到缓冲区
    full = true;          // 现在缓冲区被占用
}
```

对 `full` 的测试用来检查前置条件“缓冲区中没有单词”。

很明显，一个单词流最开始应该是空的。或者说，直到 `get()` 被第一次调用后，`full` 的值都应该是 `false`。可在单词流的定义里直接初始化成员 `full`。

6.8.2 读单词

所有的读入操作都是 `get()` 函数完成的，如果在 `Token_stream::buffer` 中没有单词，`get()` 函数必须从 `cin` 读入字符并将它们组成单词。

```
Token Token_stream::get()
{
    if (full) {                      // 缓冲区里是否已经有一个单词？
        full = false;                // 删除缓冲区里的单词
        return buffer;
    }
    char ch;                         // 注意 >> 运算符会跳过空白（空格、新行、制表符等）

    switch (ch) {
        case ';':                  // 表示“立即输出结果”
        case 'q':                  // 表示“退出”
        case '(': case ')': case '+': case '-': case '*': case '/':
            return Token{ch};        // 每个字符本身表示一个单词
        case '!':
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
        {
            cin.putback(ch);       // 将数字（或小数点）放回到标准输入流中
            double val;
            cin >> val;            // 读入一个浮点数
            return Token{'8',val};   // 用 '8' 表示“这是一个数”
        }
        default:
            error("Bad token");
    }
}
```

下面我们详细分析一下 `get()` 函数。首先检测缓冲区中是否已经有单词了，如果有就直接返回该单词：

```
if (full) {                      // 缓冲区里是否已经有一个单词？
    full = false;                // 删除缓冲区里的单词
    return buffer;
}
```

只有当 `full` 为 `false` 时（表明缓冲区中没有单词），我们才需要处理输入字符。此时，我们逐个读入字符并进行适当的处理，在其中寻找括号、运算符和数字，遇到任何其他字符我

们都将调用 `error()` 而结束程序：

```
default:
    error("Bad token");
```

`error()` 函数在 5.6.3 节中描述，我们将其声明包含在 `std_lib_facilities.h` 文件中。

我们必须考虑如何表示不同类型的单词，也就是说，必须为 `kind` 成员选择不同的值。简单起见，也为了易于调试，我们令一个单词的 `kind` 域就保存括号、运算符本身。这使得括号和运算符的处理异常简单：

```
case '(': case ')': case '+': case '-': case '*': case '/':
    return Token{ch};      // 每个字符本身表示一个单词
```

坦率地讲，我们在第 1 版中忘记了处理表示“立即输出结果”的 ‘;’ 和表示“退出”的 ‘q’ 这两个符号，我们在第 2 版中将这部分代码添加进来。

6.8.3 读数值

现在，我们必须处理数值，事实上这不是一件容易的事。如何获得 123 这个数值呢？当然，它可由 `100+20+3` 得来，但 12.34 又如何获得呢？另外，我们应该允许使用科学计数法（如 `12.34e5`）吗？为了正确实现这些功能，可能需要花几个小时甚至几天时间，幸运的是，我们可以不必做这个工作。输入流能够解析 C++ 字面常量，并能将其转换为 `double` 类型的数值。因此，我们所要做的只是如何在 `get()` 函数中告诉 `cin` 完成这些工作而已：

```
case '!':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{   cin.putback(ch);          // 将数字（或小数点）放回到标准输入流中
    double val;
    cin >> val;              // 读入一个浮点数
    return Token{'8',val};     // 用 '8' 表示“这是一个数”
}
```

某种程度上，我们是随意选择了 ‘8’ 来表示“数值”这类单词。

那么，我们如何知道输入中出现了一个数值呢？如果根据经验来推测，或者是参考 C++ 文献（如附录 A），我们会发现一个数值常量必须以一个阿拉伯数字或者小数点开头。因此，我们可以在程序中检测这些符号，来判断是否出现数值。接下来，我们希望 `cin` 完成数值的读取，但我们已经读入了第一个字符（一个阿拉伯数字或小数点）。因此，我们需要将第一个字符的数值和 `cin` 读入的后续字符的值结合起来。例如，输入 123，我们会得到 1，`cin` 读入 23，我们需要将 100 与 23 相加。这太繁琐了！幸运的是（并不是偶然的），`cin` 与 `Token_stream` 的工作方式类似，也可以把已经读出的字符放回输入流中。因此，不用做繁琐的数学运算，我们只需把第一个字符放回 `cin`，然后由 `cin` 读入整个数值。

请注意，我们如何一次又一次地避免做复杂的工作，代之以寻找简单的解决方案——通常是借助于 C++ 库。这就是程序设计的本质：不断地寻找更简单的方法。这与“优秀的程序员都是懒惰的”（看起来有些好笑？）不谋而合。从这个角度说（当然，也只有从这个角度），我们应该“懒惰”，如果能找到一个更简单的方法，我们何必写那么多代码呢？

6.9 程序结构

诗云：不识庐山真面目，只缘身在此山中。类似地，如果我们只关心一个程序中的函数、

类等的细节，就容易失去对程序的整体把握。因此，下面就忽略细节，看看程序的结构：

```
#include "std_lib_facilities.h"

class Token {/* ... */;
class Token_stream {/* ... */;

void Token_stream::putback(Token t) {/* ... */
Token Token_stream::get() {/* ... */

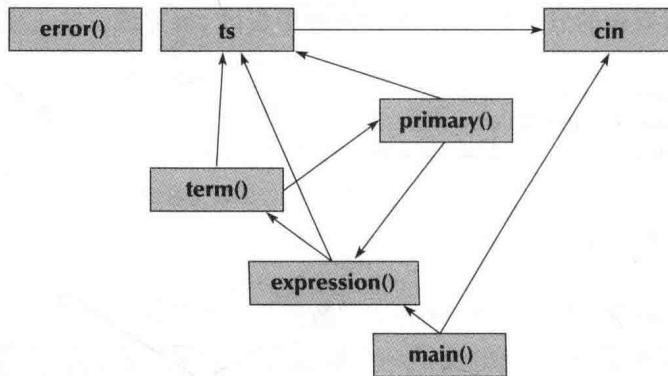
Token_stream ts;           // 提供 get() 和 putback()
double expression();       // 前置声明，这样 primary() 就可以调用 expression()

double primary() /* ... */; // 处理数值和括号
double term() /* ... */;   // 处理 * 和 /
double expression() /* ... */; // 处理 + 和 -

int main() /* ... */;      // 主循环和错误处理
```

 在这里，声明的顺序是很重要的，一个名字在被声明之前是不能使用的，因此 `ts` 必须在 `ts.get()` 使用之前声明，`error()` 必须在语法分析函数之前声明。在调用图中有一个非常有趣的循环：`expression()` 调用 `term()`，`term()` 调用 `primary()`，`primary()` 又调用了 `expression()`。

下面是调用关系的图描述（由于所有的函数都调用 `error()`，因此将其省略）：



这就意味着我们不能简单地定义这三个函数：没有任何一种顺序能满足先定义后使用的原则。因此，至少有一个函数必须先只给出声明而非定义。我们选择先声明 `expression()` 函数，这种方式称为前置声明（forward declare）。

现在的计算器程序已经能正常工作了吗？某种程度上，确实可以了。它可以正常编译、运行、正确计算表达式，并给出适当的错误信息。但它是按照我们所希望的方式工作吗？答案并不出人意料——“不全是”。6.6 节给出了程序的第一个版本并消除了一个严重的错误，6.7 节中的第二个版本并没有多少改进。但这没有关系，这是意料之中的。我们本来的目标就是写一个可以运行的程序，能用来验证我们的基本思路，从中获得反馈，对于这个目标来说现在的版本已经足够好了。从这个角度来说，它是成功的，但试一下：它仍然存在很多问题！

试一试

编译、运行上面设计的计算器程序，看看它能完成什么功能，并指出它为什么会如此工作。

简单练习

本练习对一个有很多错误的程序进行一系列改进，使其变得更加有用。

1. 编译文件 `calculator02buggy.cpp` 中的计算器程序，你需要找到并修正一些错误，才能使程序编译通过，这些错误本书中并未提及。找出并改正 `calculator02buggy.cpp` 计算器程序中存在的三个不太明显的逻辑错误，使计算器能够产生正确结果。
2. 把用于控制程序退出的命令符 `q` 换成 `x`。
3. 把用于控制程序输出的命令符 `;` 换成 `=`。
4. `ain()` 函数中增加一条欢迎信息：

```
"Welcome to our simple calculator.  
Please enter expressions using floating-point numbers."
```

5. 改进欢迎信息，提示用户可以使用哪些运算符，以及如何输出结果和退出程序。

思考题

1. “程序设计就是问题理解”的含义是什么？
2. 本章详细讲述了计算器程序的设计，简要分析计算器程序应该实现哪些功能？
3. 如何把一个大问题分解成一系列易于处理的小问题？
4. 为什么编写一个程序时，先编写一个小的、功能可控的版本是一个好主意？
5. 为什么功能蔓延是不好的？
6. 软件开发的三个主要阶段是什么？
7. 什么是“用例”？
8. 测试的目的是什么？
9. 根据本章的描述，比较 `Term`、`Expression`、`Number` 和 `Primary` 的不同点。
10. 在本章中，输入表达式被分解为 `Term`、`Expression`、`Number` 和 `Primary` 等组成部分，试按这种方式分析表达式 $(17+4)/(5-1)$ 的构成。
11. 为什么程序中没有名为 `number()` 的函数？
12. 什么是单词？
13. 什么是文法？文法规则是什么？
14. 什么是类？类的作用是什么？
15. 如何为一个类的一个成员提供一个缺省值？
16. 在 `expression()` 函数中，为什么 `switch` 语句的默认处理是退回单词？
17. 什么是“预读取”？
18. `putback()` 函数的功能是什么？为什么说它是有用的？
19. 在 `term()` 函数中，为什么难以实现取模运算符 `%`？
20. `Token` 类的两个数据成员的作用是什么？
21. 为什么把类的成员分成 `private` 和 `public` 两种类型？
22. 对于 `Token_stream` 类，当缓冲区中有一个单词时，调用 `get()` 函数会发生什么情况？
23. 在 `Token_stream` 类的 `get()` 函数中，为什么在 `switch` 语句中增加了对 ‘`;`’ 和 ‘`q`’ 的处理？
24. 应该从什么时候开始测试程序？

- 25.“用户自定义类型”是什么？我们为什么需要这种机制？
- 26.对于一个C++ 用户自定义类型，它的接口是指什么？
- 27.我们为什么要依赖代码库？

术语

analysis (分析)	grammar (文法)	prototype (函数原型)
class	implementation (实现)	pseudo code (伪代码)
class member (类成员)	interface (接口)	public
data member (数据成员)	member function (成员函数)	syntax analyzer (语法分析)
design (设计)	parser (语法分析器)	token (单词)
divide by zero (被0除)	private	use case (用例)

习题

1. 如果你尚未做本章“试一试”中的练习，请先完成。
2. 在程序中增加对0的处理，令其与0作用一致，这样， $((4+5)*6)/(3+4)$ 就是一个合法的表达式。
3. 在程序中增加对阶乘运算符（用‘!’表示）的处理，例如：表达式 $7!$ 表示 $7*6*5*4*3*2*1$ 。阶乘的优先级高于*和/，也就是说， $7*8!$ 表示 $7*(8!)$ 而不是 $(7*8)!$ 。通过修改文法来描述优先级更高的运算符，为了与数学中阶乘的定义统一，我们规定 $0!$ 等于1。提示：我们的计算器程序处理的是double型数，但阶乘只对整数有定义，所以对 $x!$ ，先将x赋给一个int型变量，再计算该int型变量的阶乘。
4. 定义包含一个字符串和一个值两个成员的类Name_value，使用vector<Name_value>替代两个vector重做第4章的习题19。
5. 将the加到6.4.1节中的“英语”文法中，以便能描述“The birds fly but the fish swim”这样的语句。
6. 根据6.4.1节中给出的“英语”文法，编写程序判断一个句子是否符合英语语法。假设每个句子都以句号(.)结束，句号的两边有空格。例如，“birds fly but the fish swim.”是一个合法的句子，但“birds fly but the fish swim”（缺少句号）和“birds fly but the fish swim.”（句号之前没有空格）都不是正确的句子。对输入的每个句子，程序能够输出“OK”或者“not OK。”提示：不需使用单词，直接使用>>来读入字符串即可。
7. 为位逻辑表达式编写一个文法。位逻辑表达式与算术表达式是非常相像的，只是前者使用的是逻辑运算符！（非）、~（补）、&（与）、|（或）和^（异或），每个运算符都对其整数操作数的每一位进行计算。其中，!与~是前缀一元运算符，^运算的优先级高于|运算（正如*运算优先级高于+运算一样），因此 $x|y^z$ 表示 $x|(y^z)$ 而不是 $(x|y)^z$ ；&运算的优先级高于^运算，因此 $x^y&z$ 表示 $x^y&z$ 。
8. 使用四个字母而不是四个数字重做第5章的习题12中的“公牛和母牛”游戏。
9. 编写一个程序，读入数字并将其组合为整数。例如，读入字符1、2、3时得到整数123，程序应输出“123 is 1 hundred and 2 tens and 3 ones”。数值由一至四个数字构成，以int类型输出。提示：如何从字符'5'得到数值5呢？可通过字符'5'减字符'0'得到，也就是说，'5'-'0'==5。

10. 一个排列是一个集合的有序子集。例如，你要从 60 个数中选取 3 个不同的数排成金库密码，则共有 $P(60, 3)$ 种排列。其中函数 P 由如下公式定义

$$P(a, b) = \frac{a!}{(a-b)!}$$

其中！是后缀阶乘运算符，例如， $4!$ 表示 $4 \times 3 \times 2 \times 1$ 。组合与排列相似，差别在于组合不关心对象的顺序。例如，如果你想从 5 种不同口味的冰淇淋中选出 3 种制作香蕉圣代，那么你不必关心香草冰淇淋是先加入的还是后加入的，因为无论如何香草冰淇淋都加进去了。组合的计算方式如下：

$$C(a, b) = \frac{P(a, b)}{b!}$$

设计一个程序，让用户输入两个数字，询问用户是计算排列还是组合，然后输出计算结果。这项工作包含以下几个步骤：首先分析上面给出的需求，确定程序需要完成的功能；然后进入设计阶段，编写伪代码，并将其分解为多个子模块。程序应该具有错误检查机制，保证程序对有问题的输入给出恰当的错误提示信息。

附言

了解输入的含义是程序设计的重要组成部分，每个程序都会以某种形式面对这个问题。其中，又以了解那些由人类直接生成的信息的含义最为困难。例如，语音识别仍然是一个非常困难的研究课题。当然，这类问题中也有一些较为简单，例如本章所研究的计算器，可以通过用文法描述输入的方式来处理。

第7章 |

Programming: Principles and Practice Using C++, Second Edition

完成一个程序

不到最后，不见分晓。

——歌剧谚语

编写程序需要不断地改进你要实现的功能及其表达方式。第6章中我们给出了一个能够正确运行的计算器程序的初步版本，本章将对其进行进一步的完善和优化。“完成程序”意味着使程序更易于用户使用，更方便开发者维护——包括改进用户接口、添加一些重要的错误处理机制、增加一些有用的功能、重构代码使之易于理解和修改。

7.1 简介

当你的程序第一次正常运行时，你大约只完成了一半工作。如果是一个大程序或者一个不正常运行会造成不良后果的程序，那连一半工作也没完成。一旦程序能初步正常运行，编程的真正乐趣就开始了！从这里开始，我们可以在初步版本之上试验各种不同的想法。

本章将引导你如何以一名专业程序员的眼光来优化第6章给出的计算器程序。值得注意的是，本章中提出的程序相关的问题以及一些思考，远比计算器程序本身有趣得多。我们将通过一个实例讲述如何在实际需求和约束条件的压力下逐步优化程序。

7.2 输入和输出

让我们回到第6章开始，你会发现我们当初决定采用提示信息 **Expression:** 提示用户输入表达式，用提示信息 **Result:** 提示输出计算结果。

迫于使程序尽快运行起来的压力，我们忽略了这些看似不重要的细节。这是很常见的，我们不可能一开始就考虑所有情况。因此，当我们停下来反思的时候，发现忘记了最初想要实现的一些功能。

对于某些程序设计任务，原始需求是不能被改变的。这一原则过于死板了，会给问题求解方案的设计带来很多困难。假定我们可以修改需求，我们又该如何做呢？应该修改哪些需求，哪些又应该保持不变？我们真的想让程序输出提示信息 **Expression:** 和 **Result:** 吗？仅仅靠“想”是不行的，最好是实际试验一下，看看哪种方式效果更好。现在我们输入

2+3; 5*7; 2+9;

输出结果为：

= 5
= 35
= 11

如果在程序中添加 **Expression:** 与 **Result:**，将得到如下结果：

Expression: 2+3; 5*7; 2+9;
Result : 5
Expression: Result: 35

Expression: **Result:** 11

Expression:

我们相信，一些人会喜欢前一种风格，而其他人会喜欢后一种。因此可以考虑允许用户选择自己喜欢的风格。但对于这个简单的计算器程序来说，提供两种输入输出风格供用户选择，过于繁琐了。因此，必须确定使用哪种风格。我们认为输出 **Expression:** 与 **Result:** 令程序有点复杂，而且会分散注意力。如果使用这些提示信息的话，真正有用的表达式输入与结果输出在屏幕显示窗口中只占很少一部分，但表达式和结果才是我们真正关心的内容，其他内容不应分散注意力。另一方面，应该把用户输入的表达式和计算机输出的结果区分开，否则用户可能会无法分辨出结果。在最初调试程序时，我们用字符 = 表示计算结果的输出。类似地，我们也可以使用一个简短的“提示符”来提示用户输入——字符 > 经常被用作用户输入提示符：

```
> 2+3;
= 5
> 5*7;
= 35
>
```

这种方式看起来好多了，我们只需对 `main()` 函数中的主循环做一点改动即可实现这种方式。

```
double val = 0;
while (cin) {
    cout << "> " ; // 打印提示符
    Token t = ts.get();
    if (t.kind == 'q') break;
    if (t.kind == ',')
        cout << "=" << val << '\n'; // 打印计算结果
    else
        ts.putback(t);
    val = expression();
}
```

不幸的是，如果在一行上输入多个表达式，其输出仍然比较混乱。

```
> 2+3; 5*7; 2+9;
= 5
> = 35
> = 11
>
```

根本原因是我们在开始开发程序时就认为用户不会在一行中输入多个表达式（至少我们假装用户不会这样）。对于这种情况，我们期望的输出方式如下：

```
> 2+3; 5*7; 2+9;
= 5
= 35
= 11
>
```

这种显示方式看起来很合理，但不幸的是，实现它却很麻烦。首先看一下 `main()` 函数，我们希望只有在后面不跟着符号 = 的情况下，才输出提示符 >，这能够实现吗？答案是否定的，因为我们根本没有办法判定这种情况！因为程序是在 `get()` 函数调用之前输出提示符 > 的，而此时无法知道 `get()` 函数是真正读取了新字符，还是简单地将已经从键盘读入的字符组成单词返回给我们。换句话说，我们可能不得不弄乱 `Token_stream` 才能实现上述输出形式。

我们认为现在的输出形式已经基本满足要求了，因此不再进行改进。如果将来我们发现必须修改 `Token_stream` 类，那时再来重新考虑这个决定。不过，修改程序的主要数据结构只是为了获得一点小小的改进，这是不明智的。而且，我们还未对计算器程序进行过全面的测试，因此目前我们决定不做输出形式上的改进。

7.3 错误处理

 当你的程序能够初步运行时，你应该做的第一件事情就是打破它，也就是给它各种输入，期望它表现出错误的行为。“期望”的意思是，在这个阶段，我们所面临的挑战是要发现尽可能多的程序错误，以便最终交付用户之前将其修正。如果你做这项工作时的态度是“我的程序已经正常运行了，我是不会犯错误的！”，那么你将不会发现很多错误，而一旦真的发现错误时，你又会非常沮丧。你应该调整自己的心态，进行程序测试时的正确态度应该是：“我能打败它！我比任何程序都聪明，即使是我自己编写的程序！”我们可以使用一些正确的和不正确的表达式混合在一起的输入来测试计算器程序，例如：

```
1+2+3+4+5+6+7+8
1-2-3-4
!+2
;;
(1+3;
(1+;
1*2/3%4+5-6;
0;
1+;
+1
1++;
1/0
1/0;
1++2;
-2;
-2;;;;
1234567890123456;
'a';
q
1+q
1+2; q
```

试一试

尝试用一些不同的“问题表达式”来测试计算器程序，看看你能使它表现出多少种不同的错误行为。你能使程序崩溃吗——使程序跳过错误处理机制而直接输出平台级的出错信息？我们认为你做不到。你能让程序异常退出而不输出任何错误信息吗？我想你是可以做到的。

这种技术称为测试 (testing)，有些人专门从事这项工作，负责找出程序中的错误。测试是软件开发中很重要的一个环节，而且并非想象的那样枯燥，实际上是可以很有趣的，我们将在第 26 章中详细讨论程序测试的一些细节问题。关于程序测试的一个重要问题是：“能否对程序进行系统测试从而发现所有的错误？”这个问题没有一个普适的答案，也就是说，没有任何一个答案对所有程序都成立。不过，对于大多数程序而言，严格的测试通常都会获得

很好的效果。程序测试最重要的环节之一是系统性地设计测试用例，为了防止测试设计不全面的情况，你可以用一些“不合理”的输入来测试程序。例如，对计算器程序输入：

```
Mary had a little lamb
srtvrqtiewcbet7rewaevre-wqcntrretewru754389652743nvcqnwq;
!@#$%^&*()~:;
```

在此，我再一次使用了我习惯性的做法：将电子邮件的内容（包括邮件头、邮件正文等所有内容）输入给编译器来测试编译器的反应。这看起来不合情理，因为“没有人会这样做”。但在实际应用中，完美的程序应该能捕获所有错误，并且能够从“奇怪的输入”中快速恢复正常运行，而不是只能处理那些“合乎情理”的错误。

在测试计算器程序时，第一个棘手的问题是当输入下列非法表达式时，程序窗口会立刻关闭。

```
+1;
0
!+2
```

稍加思考或者跟踪一下程序的执行过程就会发现，问题在于，在输出错误信息后，程序窗口就立刻关闭了。这也是我们保持窗口活跃的目的，可以等待用户输入字符后再关闭。然而，对于上述三种输入而言，程序在读入所有字符之前就检测到了一个错误，因此输入行中还剩下未被读入的字符。但是，程序不能区分它是“剩余字符”还是用户在看到提示信息 **Enter a character to close window** 后输入的字符。于是，这个“剩余字符”就被程序认为是关闭窗口的命令，导致程序窗口被关闭。

可以对 **main()** 函数稍做修改来处理这个问题（参考 5.6.3 节）：

```
catch (runtime_error& e) {
    cerr << e.what() << '\n';
    // keep_window_open();
    cout << "Please enter the character ~ to close the window\n";
    for (char ch; cin >> ch;) // 继续读入直到遇到 ~ 字符
        if (ch=='~') return 1;
    return 1;
}
```

基本上，我们将 **keep_window_open()** 函数完全替换为新的代码来处理上述问题。但需要注意，如果 ~ 恰好是发生错误之后的下一个输入字符，上述问题仍然存在，不过这种情况出现的可能性就小得多了。

我们可以编写一个新版本的 **keep_window_open()** 函数处理这个问题，它有一个字符串参数，只有用户在看到提示信息后输入这个字符串，程序才会关闭窗口，其简单实现如下：

```
catch (runtime_error& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
}
```

此时输入如下内容：

```
+1
!1~~
0
```

计算器程序会在输出错误信息后给出如下提示信息，直到用户输入 ~~ 后才退出：

Please enter ~~ to exit

计算器程序从键盘获取输入，这使得程序测试过程非常乏味：每当对程序做出改动，我们都要（再一次！）从键盘手工输入许多测试用例，以测试程序修改是否正确。因此，最好能将测试用例保存起来，用一个单独的命令就能输入这些用例进行测试。对于以 Unix 为代表的操作系统来说，在不改变程序的情况下，令 `cin` 从文件而不是键盘输入数据，令输出到 `cout` 的内容转到文件，是非常容易的。但如果在你所使用的操作系统中，这种输入输出重定向很难实现，则必须对程序代码进行修改（参见第 10 章）。

现在考虑下面两个输入

1+2; q

和

1+2 q

我们希望程序对于这两个输入都能够输出结果（3）之后退出。但奇怪的是，

1+2 q

确实是这样的，而看起来显然更正确的

1+2; q

却引发了一个 **Primary expected** 错误。我们应该如何来查找这个错误呢？在 6.7 节中，我们匆忙地在 `main()` 函数中加入了对 ; 与 q 的处理，分别表示“打印”和“退出”。现在，我们要为这种匆忙付出代价了。重新审视这部分代码：

```
double val = 0;
while (cin) {
    cout << "> ";
    Token t = ts.get();
    if (t.kind == 'q') break;
    if (t.kind == ';')
        cout << "=" << val << '\n';
    else
        ts.putback(t);
    val = expression();
}
```

在上面的代码中，判断输入是分号后不再检查 q，而是直接继续调用 `expression()` 函数。`expression()` 函数首先调用 `term()`，`term()` 又首先调用 `primary()`，由于字符 q 不是 Primary，从而输出错误信息。因此，我们应该在检测完分号以后再对字符 q 进行检测。对当前的程序，我们认为有必要适当简化程序逻辑，完整的 `main()` 函数如下：

```
int main()
try
{
    while (cin) {
        cout << "> ";
        Token t = ts.get();
        while (t.kind == ';') t=ts.get(); // 吃掉 ';' 字符
        if (t.kind == 'q') {
            keep_window_open();
            return 0;
        }
        ts.putback(t);
    }
}
```

```

        cout << "=" << expression() << '\n';
    }
    keep_window_open();
    return 0;
}
catch (exception& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open("~~");
    return 2;
}

```

改动之后的代码实现了强有力的错误处理机制，接下来我们就可以开始考虑从其他方面改进计算器程序了。

7.4 处理负数

当测试完计算器程序后，你会发现它不能很好地处理负数。例如，输入

-1/2

将返回一个错误信息，必须将其写为

(0-1)/2

但这并不符合人们的使用习惯。

在程序调试和测试后期发现这样的问题是很平常的事，只有这时我们才能有机会弄清楚程序到底实现了什么功能，并根据程序给出的反馈不断改进我们的设计。在程序的设计过程中，一种明智的做法是：在安排工作日程时就预留出时间，使我们能有机会体会开发过程中获得的经验教训，从中受益，回过头来改进程序。“1.0 版”往往未经必要的精化就发布了，这通常是由于开发日程过紧，或者是为了防止在项目“后期”对详细设计进行修改而采取的呆板的项目管理策略造成的一——“后期”添加“特性”将会是灾难性的。但实际上，当一个程序对于简单使用来说已经足够好，但还未到可以发布的程度时，开发进程还远未到“后期”。此时还是开发进程的“早期”，正是我们从程序中获取实实在在的经验教训、进行改进的好时机。在实际安排工作日程时，应该将这样的过程考虑其中。

对于本例，我们只需修改文法来处理负号。最简单的方式是修改 Primary 的定义，将

Primary:

Number
 `"(Expression)"`

改为：

Primary:

Number
 `"(Expression)"`
 `"- Primary"`
 `"+ Primary"`

我们还增加了一元加运算符，C++ 语言也支持这个运算符。当有了一元减后，人们通常也会尝试一元加，因此实现它是有意义的。而且既然实现了一元减，实现一元加也很容易，

没有必要纠缠于它到底有没有用。于是，实现 Primary 的函数变为

```
double primary()
{
    Token t = ts.get();
    switch (t.kind) {
        case '(':
            // 处理 '(' 表达式 ')'
            {
                double d = expression();
                t = ts.get();
                if (t.kind != ')') error("')' expected");
                return d;
            }
        case '8':
            return t.value; // 使用 '8' 来表示数值
        case '-':
            return -primary();
        case '+':
            return primary();
        default:
            error("primary expected");
    }
}
```

对 Primary 的修改很简单，实际上第一次就能正常工作了。

7.5 模运算 %

当我们最初分析理想中的计算器程序应该具有什么功能时，我们希望它能够处理取余（模）运算 %，但 C++ 语言中的模运算不支持浮点数，因而未加以实现。现在我们可以重新考虑模运算了，可按如下方式简单实现：

1. 添加一个新的单词（Token）%。
2. 给运算符 % 一个定义。

我们了解操作数为整数时运算符 % 的意义。例如

```
> 2%3;
= 2
> 3%2;
= 1
> 5%3;
= 2
```

但若操作数不为整数时该怎么定义呢？考虑

> 6.7%3.3;

结果应该是什么？没有一个完美的技术解决方案。但是模运算也常定义在浮点操作数上。特别是， $x \% y$ 可定义为 $x \% y = x - y * \text{int}(x/y)$ ，这样 $6.7 \% 3.3 == 6.7 - 3.3 * \text{int}(6.7/3.3)$ ，即 0.1。这可通过标准库函数 `fmod()`（浮点取模）来简单实现（见 24.8 节），需要包含头文件 `<ccmath>`。为此，在 `term()` 函数中增加以下代码：

```
case '%':
{
    double d = primary();
    if (d == 0) error("divide by zero");
    left = fmod(left, d);
    t = ts.get();
    break;
}
```

头文件 `<cmath>` 中包含了所有的标准数学函数，例如 `sqrt(x)` (`x` 的平方根), `abs(x)` (`x` 的绝对值), `log(x)` (`x` 的自然对数值) 以及 `pow(x,y)` (`x` 的 `y` 次方)。

或者我们可以禁止对浮点数进行模运算。当检测到参与模运算的浮点数有小数部分时，就给出错误提示信息。将模运算的操作数限定为整数，实际上是窄化转换（参见 3.9.2 节和 5.6.4 节）的变形之一，因此可以使用 `narrow_cast()` 函数解决：

```
case '%':
{   int i1 = narrow_cast<int>(left);
    int i2 = narrow_cast<int>(primary());
    if (i2 == 0) error("%: divide by zero");
    left = i1 % i2;
    t = ts.get();
    break;
}
```

对于一个简单的计算器程序而言，以上任何一种方案都是可以的。

7.6 清理代码

我们已经对程序做过几次修改，虽然性能每次都有所提高，但代码却变得有点乱。现在是一个很好的时机，重新检查代码，做适当的清理和简化，并增加一些注释以提高系统的可读性。换句话说，只有当代码达到易于他人接管和维护的状态，程序才算是编写完成。到目前为止，除了缺少注释外，计算器程序总体来说还是不错的，接下来我们进行一点清理工作。

7.6.1 符号常量

回忆一下，我们使用 '`8`' 表示 `Token` 中包含一个数值，这有点奇怪。实际上，采用什么值表示数值类型的单词并不重要，只要该值能够与标识其他单词类型的值区分开即可。不过，这种处理方式使得代码看起来有点古怪，我们应该使用注释语句进行相应的说明。

```
case '8':           // 使用 '8' 来表示数值
    return t.value; // 返回该数值
case '-':
    return - primary();
```

坦白说，我们也犯过一些错误，比如错敲了 '`0`' 而不是 '`8`'，因为我们忘记了到底选的是哪个值来标识数值型单词。换句话说，直接在代码中用 '`8`' 来标识数值型单词是很草率的，而且难以记忆，很容易造成人为错误——实际上 '`8`' 就是我们在 4.3.1 节中曾经提到的应该避免的“魔术常量”。我们应该为该表示数值类型单词的常量引入一个符号名：

```
const char number = '8'; // t.kind==number 表示 t 是一个数值类型单词
```

`const` 修饰符告诉编译器我们定义了一个不能被改变的对象：例如对 `number='0'`，编译器将会给出错误信息。定义了字符常量 `number` 以后，我们就不必显式地用 '`8`' 来表示数值型单词了。`primary` 函数中的相应代码片段修改如下：

```
case number:
    return t.value; // 返回该数值
case '-':
    return - primary();
```

这段代码不再需要更多注释了。实际上，代码本身直接而又清晰地表达出的内容，我们

就不应再写注释了。如果频繁地用注释来解释程序的含义，通常表明你的代码应该改进了。

类似地，`Token_stream::get()` 函数中识别数值的代码修改为：

```
case '.':
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
{   cin.putback(ch); // 将数字放回到输入流
    double val;
    cin >> val; // 读入一个浮点数
    return Token(number, val);
}
```

理论上可以为所有的单词类型设置符号名称，但是太过于繁琐。毕竟，用 '`'` 和 '`+`' 表示左括号和加号这两个单词，是任何人都能够理解的很显然的表示方法。检查计算器程序涉及的单词，发现只有用 '`;`' 表示“打印”（或“表达式结束”）以及用 '`'q'` 表示“退出”有些不妥。为什么不用 '`'p'` 和 '`'e'`' 呢？在一个大程序中，这种模糊而随意的表示方式迟早会引起问题，所以我们引入如下声明：

```
const char quit = 'q'; // t.kind==quit 意指 t 是一个表示退出的单词
const char print = ';'; // t.kind==print 意指 t 是一个表示打印的单词
```

下面修改 `main()` 函数中的循环代码：

```
while (cin) {
    cout << "> ";
    Token t = ts.get();
    while (t.kind == print) t = ts.get();
    if (t.kind == quit) {
        keep_window_open();
        return 0;
    }
    ts.putback(t);
    cout << "=" << expression() << "\n";
}
```

引入符号名称“`print`”和“`quit`”后，提高了代码的可读性。另外，我们并不鼓励人们通过阅读 `main()` 函数来推测输入什么内容表示“`print`”和“`quit`”。例如，如果我们决定用 '`e`' (代表 `exit`) 表示“`quit`”，应该是很正常的，而且这一改动应该不用改变 `main()` 函数中的任何代码。

输入 / 输出提示符 '`>`' 和 '`=`' 也同样存在问题，代码中存在这么多概念模糊的符号，如何让初级程序员在阅读 `main()` 函数时猜测其正确含义？为代码添加注释是一个好主意，但如前所述，引入符号常量更加有效：

```
const string prompt = "> ";
const string result = "= "; // 用来表示接下来的是输出结果
```

我们想改变输入 / 输出提示符时该怎么办呢？只需修改这些常量即可，主函数中的循环修改如下：

```
while (cin) {
    cout << prompt;
    Token t = ts.get();
    while (t.kind == print) t = ts.get();
    if (t.kind == quit) {
        keep_window_open();
        return 0;
    }
```

```

    }
    ts.putback(t);
    cout << result << expression() << '\n';
}

```

7.6.2 使用函数

程序中所使用的函数应该反映出该程序的基本结构，而函数名则应有效地标识代码的逻辑独立部分。到目前为止，计算器程序在这些方面做得还是比较好的：`expression()`、`term()`和`primary()`直接反映出我们对表达式文法的理解，而函数`get()`则用来处理表达式输入和单词识别。分析一下主函数`main()`，我们注意到它主要做了两项逻辑上相互独立的任务。

1. `main()` 函数搭起了程序的整体框架：启动程序、结束程序、处理致命错误。
2. `main()` 函数用一个循环来计算表达式。

理想情况下，一个函数只实现一个独立的逻辑功能（参见 4.5.1 节）。而`main()`实现了两个功能，这使得程序结构变得有些模糊。一种显然的改进方法是将表达式计算循环从主函数中分离出来，实现为`calculate()`函数：

```

void calculate() // 表达式计算循环
{
    while (cin) {
        cout << prompt;
        Token t = ts.get();
        while (t.kind == print) t=ts.get(); // 先丢弃所有的“打印”单词
        if (t.kind == quit) return;
        ts.putback(t);
        cout << result << expression() << '\n';
    }
}

int main()
try {
    calculate();
    keep_window_open(); // 处理 Windows 控制台模式
    return 0;
}
catch (runtime_error& e) {
    cerr << e.what() << '\n';
    keep_window_open("~~");
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open("~~");
    return 2;
}

```

修改后的代码更直接地反映了程序的结构，更易于理解。

7.6.3 代码布局

重新检查一下计算器程序，看看其中是否有“丑陋”的代码，我们发现：

```

switch (ch) {
case 'q': case ',': case '%': case '(': case ')': case '+': case '-': case '*': case '/':
    return Token{ch}; // 用每个字符本身代表它对应的单词类型
}

```

在加入对 'q'、';' 和 '%' 的处理之前，这段代码还不算太坏，但现在变得有些混乱。代码的可读性越差，其中的错误就越难以发现。而这段代码中确实隐藏着一个潜在的 bug！我们修改一下代码，令每行代码只对应 `switch` 语句的一种情况，并加入适当的注释来帮助代码理解。修改后的 `Token_stream::get()` 函数如下所示：

```
Token Token_stream::get()
    // 从 cin 读入字符并构成一个单词
{
    if (full) { // 检查是否已经有一个单词
        full = false;
        return buffer;
    }
    char ch;
    cin >> ch; // 注意 >> 跳过了所有空白（空格、换行、制表符等）

    switch (ch) {
        case quit:
        case print:
        case '(':
        case ')':
        case '+':
        case '-':
        case '*':
        case '/':
        case '%':
            return Token{ch}; // 用每个字符本身代表它对应的单词类型
        case '.':
            // 浮点数可由 '.' 开始
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9': // 数字
        {
            cin.putback(ch); // 将数字放回到输入流
            double val;
            cin >> val; // 读入一个浮点数
            return Token{number, val};
        }
        default:
            error("Bad token");
    }
}
```

我们当然可以把对每个数字的处理也放在不同的行，但是那样似乎并不能使代码更加清晰，而且导致不能在一屏上显示 `get()` 函数的所有代码。理想情况是，每个函数的代码都能全部显示在屏幕的可视区域上——在屏幕之外我们无法看到的代码是最有可能隐藏 bug 的地方。因此，代码布局是非常重要的。

另外一个值得注意的地方是，我们在程序中用符号常量 `quit` 替代了字符 '`q`'。这不但提高了程序的可读性，而且保证我们的编程错误会被编译器捕获——如果我们为 `quit` 操作选择的单词与其他单词冲突，将会产生一个编译时错误。

 在代码清理阶段，我们有可能意外地引入一些错误。因此，在代码清理之后一定要测试代码的正确性。最好是每做一点改动就测试一次，以便发现错误时你能记起来是做了什么样的改动导致的这个错误。记住：及早测试，经常测试。

7.6.4 注释

 我们在编写计算器程序的过程中加入了一些注释。好的注释是程序代码的重要组成部分

分。在程序开发进度很紧时，我们往往会忽略注释。当我们回过头来进行代码清理的时候，是一个很好的时机来全面检查程序的每个部分，检查原来所写的注释是否满足以下要求：

1. 在改动了程序代码以后，原来的注释是否仍然有效？
2. 对读者来说注释是否充分？（通常是不够充分的。）
3. 是否简短清晰，不至于分散读者看代码的注意力？

强调一下最后一条：最好的注释就是让程序本身来表达。如果读者了解程序设计语言，对一些意义已经很明确的代码，就应该避免不必要的冗长注释。例如：

```
x = b+c; // 将 b 和 c 相加，并将结果赋给 x
```

你可能会在本书中发现一些类似的注释，但只限于用来解释你所不熟悉的语言特性的用法。

注释一般用于代码本身很难表达思想的情况。换句话说，代码说明它做了什么，但没有表达出它做这些的目的是什么（参见 5.9.1 节）。回顾一下计算器程序，其中就缺少一些必要的注释：函数本身说明了我们是如何处理表达式和单词的，但没有给出表达式和单词的具体含义。对于计算器程序，表达式的文法最适合放入代码注释或者说明文档中，以此解释表达式和单词的含义。

```
/*
简单计算器程序

修订历史：

Revised by Bjarne Stroustrup November 2013
Revised by Bjarne Stroustrup May 2007
Revised by Bjarne Stroustrup August 2006
Revised by Bjarne Stroustrup August 2004
Originally written by Bjarne Stroustrup
(bs@cs.tamu.edu) Spring 2004.

本程序实现了一个简单的表达式计算器。
从 cin 读入；输出到 cout。
输入文法如下：
Statement:
    Expression
    Print
    Quit

Print:
;

Quit:
q

Expression:
    Term
    Expression + Term
    Expression - Term

Term:
    Primary
    Term * Primary
    Term / Primary
    Term % Primary

Primary:
    Number
```

```
( Expression )
- Primary
+ Primary
Number:
floating-point-literal
```

通过名为 `ts` 的单词流从 `cin` 输入
*/

我们这里使用了块注释，它以 `/*` 开头，一直到 `*/` 结束。注释的开始是程序的版本变化历史，在实际程序中，版本历史一般用于记录每个版本相对于上个版本做了哪些修正和改进。

注意，注释不是代码。实际上，上面注释中的文法已经进行了简化：对比注释中 `Statement` 的规则和实际的程序实现可以看出来（参见下一节中的代码）。注释中的规则无法说明 `calculate()` 中的循环语句可以在一次程序执行中计算多个表达式的情况。我们在 7.8.1 节中将对这个问题进行进一步探讨。

7.7 错误恢复

为什么程序遇到错误就结束运行呢？当初我们选择策略时，这种方式确实看起来简单明了。但是，为什么不让程序给出一个错误提示信息，然后继续运行呢？毕竟，我们常常会出一些小的输入错误，而这并不意味着我们打算结束程序的运行。因此，我们下面尝试为程序加入错误恢复能力。这意味着，程序必须能够捕获异常，并在清理遗留故障后继续运行。

在现在的计算器程序中，所有错误都表示为异常，由 `main()` 函数处理。如果我们希望加入错误恢复功能，必须让 `calculate()` 函数捕获异常，并在计算下一个表达式之前清理故障。

```
void calculate()
{
    while (cin)
        try {
            cout << prompt;
            Token t = ts.get();
            while (t.kind == print) t=ts.get(); // 先丢弃所有的“打印”单词
            if (t.kind == quit) return;
            ts.putback(t);
            cout << result << expression() << '\n';
        }
        catch (exception& e) {
            cerr << e.what() << '\n'; // 输出错误信息
            clean_up_mess();
        }
}
```

我们简单地将 `while` 循环代码块放在 `try` 代码块中，`try` 代码块在捕获异常后给出错误提示信息，并清理遗留故障。在此之后，程序如往常一样继续运行。

“清理遗留故障”的必要性何在？本质上来说，在错误处理之后准备好继续进行下面的运算，就意味着与错误相关的程序数据都已清理，所有数据都已处于良好的、可预测的状态。在计算器程序中，`Token_stream` 是唯一在函数之外定义的数据。因此，我们所要做的就是清理与错误表达式相关的所有单词，避免它们弄乱下一个表达式。例如：

`1++2*3; 4+5;`

将会引发一个错误，即第二个`+`触发异常之后，`Token_stream` 和 `cin` 的缓冲区中仍然保存着`2*3; 4+5;`。对此有两种处理方式：

1. 清除 `Token_stream` 中的所有单词。
2. 清除 `Token_stream` 中与当前表达式相关的所有单词。

第一种方式将清除所有单词（包括`4+5;`），而第二种方式只清除`2*3;`，留下`4+5`，随后将被计算。看起来哪种选择都是合理的，但都会让某些用户感到奇怪。实际上，两种方式的实现都比较简单，为了简化测试，我们选择第二种处理方式。

只需在碰到分号之前一直由 `get()` 函数读取输入，并编写如下所示的 `clean_up_mess()` 函数：

```
void clean_up_mess()           // 有问题
{
    while (true) {            // 跳过，直至找到“打印”单词
        Token t = ts.get();
        if (t.kind == print) return;
    }
}
```

不幸的是，这种方式并不能处理所有的情况。考虑如下输入：

```
1@z; 1+3;
```

@ 会导致程序执行 `catch` 子句，进而调用 `clean_up_mess()` 函数查找下一个分号，然后 `clean_up_mess()` 函数调用 `get()` 函数读入字符 `z`。由于 `z` 不是预定义的单词，这引起了另一个错误，程序转到 `main()` 函数中的 `catch(...)` 块，最后导致程序结束。太糟糕了！我们还没有机会计算表达式`1+3`的值，程序就退出了。我们只能从头开始！

我们可以尝试更精巧的 `try` 和 `catch` 结构，但这样会使程序更加混乱。错误处理是程序设计中比较困难的部分，而错误处理过程中发生的错误就更难处理。因此，我们需要设计一种不用抛出异常就能够从 `Token_stream` 中清除字符的方法。在计算器程序中，`get()` 函数是获取输入的唯一途径，但如我们刚刚所见，它遇到错误会抛出一个异常。因此，我们需要设计一个新的操作，很明显，应该将它放在 `Token_stream` 中：

```
class Token_stream {
public:
    Token get();           // 获取一个单词
    void putback(Token t); // 放回一个单词
    void ignore(char c);  // 忽略直到字符 c 的所有字符 (c 也忽略)
private:
    bool full {false};    // 缓冲区里是否有单词？
    Token buffer;         // 这是我们存储通过 putback() 放回的单词的缓冲区
};
```

由于 `ignore()` 函数需要检查 `Token_stream` 的缓冲区，因此将其定义为 `Token_stream` 的一个成员函数。我们将“希望找到的内容”作为函数 `ignore()` 的参数，因为毕竟哪些字符可用于错误恢复是上层程序的事情，`Token_stream` 无须了解。我们将这个参数设置为一个字符，是因为希望按原始字符来处理输入，错误恢复过程中提取单词的缺点在前面已经看到了。因此有

```
void Token_stream::ignore(char c)
// c 代表单词类型
{
    // 首先查看缓冲区
    if (full && c==buffer.kind) {
```

```

    full = false;
    return;
}
full = false;

// 现在查找输入流
char ch = 0;
while (cin >> ch)
    if (ch == c) return;
}

```

程序首先检查缓冲区，如果缓冲区中的字符就是 c，则丢掉它，结束函数；否则一直从 cin 中读入字符，直到遇到 c 为止。

现在，clean_up_mess() 函数可以简化为：

```

void clean_up_mess()
{
    ts.ignore(print);
}

```

错误处理通常是比较困难的。由于很难猜测程序到底会出现什么样的错误，因此需要进行大量的实验与测试。编写一个万无一失的程序对技术要求很高，业余程序员通常会忽略这一方面，因此高质量的错误处理往往是衡量程序员专业程度的标志。

7.8 变量

我们已经对程序风格和错误处理机制进行了改进，接下来该回过头进一步完善程序的功能了。我们现在已经有一个运行得相当好的程序了，该如何完善它呢？我们希望加入的第一个功能是对变量的支持，变量的加入能够令使用者更好地表示长表达式。类似地，对于科学计算，我们希望支持内置的命名常量，如 pi 和 e 等，就像大多数科学计算器那样。

增加变量和常量是对计算器程序的重大扩展，会涉及程序的大部分代码，如果没有充分的理由和时间，最好不要进行这种类型的修改。我们这里为计算器程序增加变量和常量的功能，是因为一方面我们可以借此重新检查程序代码，另一方面还可以学习更多的程序设计技巧。

7.8.1 变量和定义

很明显，对于变量和内置常量来说，最关键的是保存 (name, value) 对，从而通过名字来访问相应的值。可以定义 Variable 类如下：

```

class Variable {
public:
    string name;
    double value;
};

```

于是我们就可以使用 name 成员来标识一个 Variable，用 value 成员存储与 name 对应的值。

我们应该如何存储 Variable 对象，以便能根据 name 来查找 Variable 并存取对应的值？回顾一下我们目前所学的程序设计工具，最好的方法是使用 Variable 的 vector：

```
vector<Variable> var_table;
```

我们可以在 var_table 中存放任意多个 Variable 对象，当搜索某个给定的 name 时，只要顺序

查找 `vector` 的每个元素即可。我们可以编写一个函数 `get_value()` 来实现查找给定 `name`，并返回对应的值：

```
double get_value(string s)
    // 返回 name 为字符串 s 的 Variable 对应的值
{
    for (const Variable& v : var_table)
        if (v.name == s) return v.value;
        error("get: undefined variable ", s);
}
```

函数代码很简单：遍历 `var_table` 中的每个 `Variable` 对象（从 `vector` 的第一个元素开始，逐个元素访问，直至最后一个元素），判断变量的 `name` 成员是否与字符串参数 `s` 匹配。若匹配，则返回 `value` 成员中的值。

类似地，我们还可以定义 `set_value()` 函数实现对 `Variable` 的赋值：

```
void set_value(string s, double d)
    // 设置 name 为字符串 s 的 Variable 对应的值为 d
{
    for (Variable& v : var_table)
        if (v.name == s) {
            v.value = d;
            return;
        }
    error("set: undefined variable ", s);
}
```

现在可以读写 `var_table` 中已存在的变量（描述为 `Variable`）了，但是如何向 `var_table` 增加一个新的 `Variable` 呢？计算器程序的使用者应该输入什么内容来定义一个新的变量，以便随后使用它呢？我们可以考虑采用 C++ 的语法：

```
double var = 7.2;
```

这样是可行的，但计算器程序中的所有变量都是 `double` 类型的，所以这里的“`double`”是可以省略的，变量的定义可简化为：

```
var = 7.2;
```

但是，有时候不能正确区分是变量定义还是书写错误。例如：

```
var1 = 7.2; // 定义一个新变量 var1
var1 = 3.2; // 定义一个新变量 var2
```

很明显，我们的原意是 `var2=3.2;`，但输入发生了错误（注释没有输入错）。对于这种输入错误导致混淆的情况，我们可以接受它，但更好的方式是遵循程序设计语言（如 C++）的习惯，区分变量的声明（包括初始化）和赋值操作。我们可以使用 `double`，但是在计算器程序中，我们选择更短的关键字 `let` 来定义变量——它实际上来自更老的传统习惯。

```
let var = 7.2;
```

相关文法如下：

```
Calculation:
Statement
Print
Quit
Calculation Statement
```

Statement:
Declaration
Expression

Declaration:
"let" Name "=" Expression

Calculation 是文法中新增的顶层产生式（规则），表示 `calculate()` 函数中的循环可以在计算器程序的一次运行过程中执行多次计算。它依赖 **Statement** 产生式处理表达式和声明。处理 **Statement** 规则的函数如下：

```
double statement()
{
    Token t = ts.get();
    switch (t.kind) {
        case let:
            return declaration();
        default:
            ts.putback(t);
            return expression();
    }
}
```

现在可以用 `statement()` 替代 `calculate()` 中的 `expression()`：

```
void calculate()
{
    while (cin)
        try {
            cout << prompt;
            Token t = ts.get();
            while (t.kind == print) t=ts.get(); // 先丢弃所有的“打印”单词
            if (t.kind == quit) return; // 退出
            ts.putback(t);
            cout << result << statement() << '\n';
        }
        catch (exception& e) {
            cerr << e.what() << '\n'; // 输出错误信息
            clean_up_mess();
        }
}
```

现在我们必须编写 `declaration()` 函数，它应该完成什么功能？应该保证在 `let` 之后出现的是一个 `Name` 接一个 `=` 再接一个 `Expression`——语法所描述的形式。应该对 `name` 做何处理？我们应该向 `var_table` 中加入一个 `Variable` 对象，其中两个成员分别设置为 `name` 的字符串内容和 `expression` 的值。在随后的表达式计算过程中，我们就可以使用 `get_value()` 和 `set_value()` 函数对变量值进行读写。然而，在动手编写代码之前，我们应该考虑一个问题——如何处理一个变量定义两次的情况？例如：

```
let v1 = 7;
let v1 = 8;
```

一般把这种重复定义作为错误来处理，实际中这往往是拼写错误所致。如本例，我们实际上是想定义两个变量：

```
let v1 = 7;
let v2 = 8;
```

使用变量名 var 及值 val 定义一个 Variable，从逻辑上应该分成两个部分：

1. 检查 var_table 中是否已经存在名为 var 的 Variable。
2. 将 (var, val) 加到 var_table 中。

这里不支持未初始化的变量。我们定义函数 is_declared() 和 define_name() 分别实现上述两个独立的逻辑操作：

```
bool is_declared(string var)
    // var 是否已经在 var_table 中?
{
    for (const Variable& v : var_table)
        if (v.name == var) return true;
    return false;
}
double define_name(string var, double val)
    // 将 (var, val) 加入 var_table 中
{
    if (is_declared(var)) error(var, " declared twice");
    var_table.push_back(Variable(var, val));
    return val;
}
```

可以通过 vector 的成员函数 push_back() 将一个新的 Variable 添加到 `vector<Variable>` 中：

```
var_table.push_back(Variable(var, val));
```

其中，`Variable(var, val)` 创建相应的 Variable 对象，然后 `push_back()` 将它添加到 `var_table` 的末尾。假设我们已经能够处理 let 和 name 单词，可以直接得到函数 declaration() 的实现：

```
double declaration()
    // 假设已经看到了 "let"
    // 处理：name = expression
    // 声明一个变量，以 "name" 命名，初始值为 "expression"
{
    Token t = ts.get();
    if (t.kind != name) error("name expected in declaration");
    string var_name = t.name;

    Token t2 = ts.get();
    if (t2.kind != '=') error("= missing in declaration of ", var_name);

    double d = expression();
    define_name(var_name, d);
    return d;
}
```

注意，我们在函数末尾返回了新变量的值，当初始化表达式比较复杂时这种方式是比较有用的，例如：

```
let v = d/(t2-t1);
```

这个声明语句定义了变量 v 并打印它的值。打印声明变量的值简化了 calculate() 函数的代码，因为这样的话每个 statement() 函数都会返回一个值。一般化原则会使代码简单，而特殊情况会使问题变得复杂。

这种跟踪变量的机制通常被称为符号表 (symbol table)，如果使用标准库中的 map 的话，这部分代码会得到极大的简化 (参见 16.6.1 节)。

7.8.2 引入 name 单词

到现在为止，我们已经对程序进行了很好的改进，但很遗憾，它还不能正常运行。这并不意外，我们对程序下的“第一刀”通常是不会正常工作的，因为我们甚至还没有完成程序——程序还无法通过编译。程序还不能识别单词 `=’，但这可以通过在 `Token_stream::get()` 中添加一种情况处理来简单实现。但是对于单词 `let` 和 `name`，必须修改 `get()` 函数来识别这些单词。一种实现如下：

```
const char name = 'a';           // name 单词
const char let = 'L';            // 声明单词
const string declkey = "let";    // 声明关键字

Token Token_stream::get()
{
    if (full) {
        full = false;
        return buffer;
    }
    char ch;
    cin >> ch;
    switch (ch) {
        // 如前
    default:
        if (isalpha(ch)) {
            cin.putback(ch);
            string s;
            cin>>s;
            if (s == declkey) return Token(let); // 声明关键字
            return Token{name,s};
        }
        error("Bad token");
    }
}
```

首先请注意函数调用 `isalpha(ch)`，它用来检测输入 `ch` 是否为字符。`isalpha()` 是一个标准库函数，可通过包含头文件 `std_lib_facilities.h` 来使用。更多的字符分类函数的内容可以参考 11.6 节。识别变量名与识别数字的方法是相同的：找到一个正确类别的字符（这里是一个字母）以后，使用 `putback()` 函数把它退回，然后使用 `>>` 读取整个变量名。

不幸的是，程序还是不能通过编译，因为 `Token` 无法存储一个字符串，编译器不能识别 `Token{name, s}`。不过，修改 `Token` 的定义可以解决这个问题，必须使 `Token` 可以存储一个 `string` 或者 `double`，并支持三种不同的初始化方法，即

- 只有 `kind`，例如 `Token{`*`}`。
- 一个 `kind` 和一个数，例如 `Token{number, 4.321}`。
- 一个 `kind` 和一个 `name`，例如 `Token{name, "pi"}`。

为此，引入三个初始化函数，由于它们是用于构造对象的，所以我们称之为构造函数：

```
class Token {
public:
    char kind;
    double value;
    string name;
```

```

Token(char ch) :kind{ch} {}           // 将 kind 初始化为 ch
Token(char ch, double val) :kind{ch}, value{val} {} // 初始化 kind 和 value
Token(char ch, string n) :kind{ch}, name{n} {}      // 初始化 kind 和 name
};

```

构造函数可使初始化过程更灵活，也易于控制。关于构造函数的细节将在第 9 章（9.4.2 节和 9.7 节）中展开。

这里用字符 'L' 表示单词 let，字符串 let 作为关键字。很明显，将关键字改为 double、var、# 或者其他任何字符串都是很容易的，只要将 declkey 的值改为想要的字符串，get() 函数中读入名字 s 后与 declkey 比较，就能识别出相应的关键字。

现在重新运行程序，如果输入下列表达式程序将能够正常运行：

```

let x = 3.4;
let y = 2;
x + y * 2;

```

但是，程序不能正确计算下列表达式：

```

let x = 3.4;
let y = 2;
x+y*2;

```

两个例子有什么差别吗？让我们仔细检查一下发生了什么情况。

问题在于我们定义 Name 时太马虎了，甚至“忘记”去定义 Name 的产生式（参见 7.8.1 节）。根据现有的文法，什么样的字符可以作为名字的一部分呢？字母？当然可以。数字呢？也可以，只要不是首字符就可以。那么下划线呢？“+”呢？应该是不允许的，我们的程序没有正确处理它们。我们还是回过头来认真检查一下代码吧。当读入首字母以后，我们用 >> 读入一个字符串。这会把空格以前的所有字符都读取到字符串中。也就是说，x+y*2；虽然是一个表达式，但这里却作为一个变量名处理，甚至分号也成了变量名的一部分。这显然不是我们的本意，也是无法接受的。

应该如何修正这个错误呢？首先，我们必须精确地定义 name 是什么；然后，我们必须修改 get() 函数来实现 name 的读取。一个可行的 name 的定义是以字母开头的字母 / 数字串，则下面的字符串都是 name；

```

a
ab
a1
Z12
asdsddsfdfdasfda434RTHTD12345dfdsa8fsd888fadsf

```

而下面的字符串都不是：

```

1a
as_s
#
as*
a car

```

当然，按 C++ 的语法，as_s 是一个合法的 name。可将 get() 函数的默认情形修改如下：

```

default:
    if (isalpha(ch)) {
        string s;
        s += ch;
        while (cin.get(ch) && (isalpha(ch) || isdigit(ch))) s+=ch;
    }
}

```

```

    cin.putback(ch);
    if (s == declkey) return Token{let}; // 声明关键字
    return Token{name,s};
}
error("Bad token");

```

新的代码把原来直接将字符串读入 s 的方式改为不断读入字符，只要是字母或者数字，就添加到 s 的末尾（语句 `s+=ch` 表示将字符 ch 添加到字符串 s 的末尾）。while 语句看起来很奇怪：

```
while (cin.get(ch) && (isalpha(ch) || isdigit(ch))) s+=ch;
```

这条语句中使用 `cin` 的成员函数 `get()` 读一个字符到 `ch`，并检查是否为字母或者数字。如果是，就将 `ch` 添加到 `s` 的末尾，然后继续读入下一个字符。成员函数 `get()` 与 `>>` 的作用相似，只是它缺省情况下不会跳过空格。

7.8.3 预定义名字

现在程序已经支持命名变量了，我们可以预定义一些常用的名字。例如：假如计算器程序用于科学计算，那么我们可能需要预定义 `pi` 和 `e`。我们应该在程序中什么位置放置这些定义？可以放在 `main()` 函数中的 `calculate()` 函数调用以前，或者放在 `calculate()` 函数中的计算循环之前。由于这些定义不是任何表达式计算的组成部分，因此可以将它们放在 `main()` 函数中。

```

int main()
try {
    // 预定义名字
    define_name("pi",3.1415926535);
    define_name("e",2.7182818284);

    calculate();

    keep_window_open();      // 处理 Windows 的控制台模式
    return 0;
}
catch (exception& e) {
    cerr << e.what() << '\n';
    keep_window_open("~/");
    return 1;
}
catch (...) {
    cerr << "exception \n";
    keep_window_open("~/");
    return 2;
}

```

7.8.4 我们到达目的地了吗

还没有，在对程序做了诸多修改以后，还需要对程序进行测试、清理代码和修改注释等。而且，还可以定义更多的操作和变量。例如：我们还没有在程序中提供赋值操作符（见习题 2），如果实现赋值操作的话，我们可能还想区分变量和常量（见习题 3）。

我们先回到最初不支持命名变量的计算器程序，仔细回顾一下实现命名变量功能的代码，可能会有两种不同的反应：

1. 实现变量并不那么糟，大概用三四十行代码就可以了。

2. 实现变量是一个重大的扩展，几乎涉及每个函数，并且在计算器程序中引入了一个全新的概念。在没有实现赋值操作的情况下，代码量已经增加了大约 45%！

计算器程序是我们第一个比较复杂的程序，站在这个角度来看，第二种反应是比较合理的。一般来说，如果一个改进程序的建议会使程序的代码量和复杂度都增加 50% 左右，第二个反应是很正常的。如果真按这样的建议去做了，你会发现整个过程更像是基于原来版本重写了一个新的程序。而且，你最好把这个过程当作重写程序来对待，这样会有更好的效果。特别地，如果我们能够分阶段编写和测试程序，就像设计计算器程序这样，最好就这么做，这比一下子就编写完整的程序要好得多。

简单练习

1. 从程序 `calculator08buggy.cpp` 开始，修改其中的错误，使之能编译通过。
2. 阅读整个计算器程序并添加适当的注释。
3. 在注释的过程中，你会发现程序中存在一些错误（我们特意加入了一些不明显的错误让你来查找），这些错误都未在本章中出现过，尝试修正它们。
4. 测试：准备一组测试数据，用来测试计算器程序。要注意测试用例的完整性，思考你要通过测试用例查找什么。测试用例应包括负数、0、非常小的数、非常大的数和一些“愚蠢”输入。
5. 进行测试，并修改在注释代码过程中没有发现的错误。
6. 增加一个预定义名字 `k`，其值为 1000。
7. 为用户提供平方根函数 `sqrt()`，比如允许用户计算 `sqrt(2+6.7)`。`sqrt(x)` 的值是 `x` 的平方根，例如 `sqrt(9)` 的值为 3。使用标准库函数 `sqrt()` 完成平方根的计算，这个函数包含在头文件 `std_lib_facilities.h` 中。记得更新程序代码注释以及文法。
8. 捕获求负数的平方根的异常，并给出适当的错误信息。
9. 增加幂函数 `pow(x,i)`，例如 `pow(2.5, 3)` 表示 $2.5 \times 2.5 \times 2.5$ ，要求 `i` 为整数，可使用与 % 运算符相同的方法处理。
10. 将“声明关键字”`let` 改为 `#`。
11. 将“退出关键字”`quit` 改为 `exit`，与 `let` 的处理一样，我们需要定义一个表示“退出”的字符串，参见 7.8.2 节。

思考题

1. 为什么还要对程序的第一版本做这些改进？给出几条原因。
2. 为什么输入表达式 `1+2; q` 后，程序没有退出而是给出一个错误信息？
3. 为什么选择把一个字符常量叫作 `number`？
4. 为什么把 `main()` 函数分成两个相互独立的部分，分别实现了什么功能？
5. 为什么把程序代码分成若干个小函数？试阐明划分原则。
6. 代码注释的目的是什么？如何为程序增加注释？
7. `narrow_cast` 的作用是什么？
8. 符号常量的使用方法是什么？
9. 为什么关心代码布局？
10. 如何处理浮点数的模运算（%）？

11. `is_declared()` 函数的功能是什么？它是如何工作的？
12. `let` 单词对应的输入内容是由多个字符构成的，在修改后的程序中，如何将其作为单个单词读入？
13. 计算器程序中的变量名可以是什么形式，不能是什么形式，对应的规则是怎样的？
14. 为什么说以增量方式设计程序是一个比较好的主意？
15. 什么时候开始对程序进行测试？
16. 什么时候对程序进行再测试？
17. 如何决定函数的划分？
18. 如何为变量和函数起名字？列出一些可能的理由。
19. 为什么添加代码注释？
20. 注释中应该写些什么内容，什么内容不应该写？
21. 什么时候可以认为已经完成了一个程序？

术语

code layout (代码布局)	maintenance (维护)	scaffolding (程序框架)
commenting (注释)	recovery (恢复)	symbolic constant(符号常量)
error handling (错误处理)	revision history (版本历史)	testing (测试)
feature creep (功能蔓延)		

习题

1. 修改计算器程序，允许名字中出现下划线。
2. 提供赋值操作符 `=`，以便在 `let` 定义变量后能够修改其值。试讨论赋值操作符的用处以及可能引起的问题。
3. 提供命名常量——不允许更改其值。提示：在 `Variable` 中增加一个用于区分常量和变量的成员，并根据该成员判断 `set_value()` 是否允许执行。如果允许用户定义常量（而不是计算器程序中预定义的 `pi` 和 `e` 那样的常量），必须增加一个符号，用户可用其表达常量定义，例如用 `const` 表示常量定义：`const pi=3.14`。
4. `get_value()`、`set_value()`、`is_declared()` 和 `declare_name()` 等函数都操作全局变量 `var_table`。定义一个 `Symbol_table` 类，其中一个成员是 `vector<Variable>` 类型的 `var_table`，成员函数为 `get()`、`set()`、`is_declared()` 和 `declare()`，并使用该类重新编写计算器程序。
5. 修改 `Token_stream::get()` 函数，在读到换行时返回单词 `print`。这就需要寻找空格并对换行 (`\n`) 进行特殊处理。可以使用标准库函数 `isspace(ch)`，当 `ch` 为空格时返回 `true`。
6. 每个程序都应该具有给用户提供帮助信息的功能，修改计算器程序，当用户输入 `H`（大小写均可）时能够显示帮助信息。
7. 将命令符 `q` 和 `h` 分别改为 `quit` 和 `help`。
8. 7.6.4 节给出的文法是不完整的（我们提醒过你不要过分依赖注释），没有定义语句序列，例如 `4+4; 5-6;`，也不包括 7.8 节对文法所做的改进。修改文法，并为注释添加你认为必要的内容。
9. 为计算器程序提出三种本章没有提及的改进，并实现其中的一种。
10. 修改计算器程序，使其仅仅能处理整数，并在发生上溢和下溢时给出错误信息。提示：

利用 `narrow_cast` (见 7.5 节)。

11. 回顾你在做第 4 章和第 5 章习题时所编写的两个程序，根据本章给出的原则清理代码，看看在这个过程中能否发现错误。

附言

很偶然地，我们通过一个简单的例子了解了编译器是如何工作的。计算器程序分析输入流，将其分解为单词，并根据文法规则理解其意义。这正是编译器所做的工作。在分析了前面阶段的输出之后，编译器将生成另外一种形式的描述（目标代码），可供随后执行。而计算器程序分析了表达式的含义后，会立刻计算其值，这种程序一般被称为解释器，而不是编译器。

函数相关的技术细节

再多的天赋也战胜不了对细节的偏执。

——俗语

在本章和下一章中，我们将注意力从程序设计转移到主要的编程工具——C++ 上。我们会介绍一些语言的技术细节，给出一个 C++ 的基本功能的稍宽的视角，并从更系统化的角度讨论这些功能。这两章还会回顾很多前面章节介绍过的程序设计概念，并提供一个研究语言工具的机会，这两章不介绍新的程序设计技术和概念。

8.1 技术细节

如果可以选择的话，我们更愿意讨论程序设计，而不是程序设计语言的特性；也就是说，我们认为，如何用代码表达思想远比我们用来表达这些思想的程序设计语言技术细节更有意思。自然语言中有类似的情况：我们更愿意讨论一部好的小说中的思想和它表达这些思想的方式，而不会愿意研究其中的英语语法和词汇。总之，重要的是思想和如何用代码表达思想，而不是单独的编程语言特性。

但是，我们不是总能选择。当你开始编程时，你用的编程语言对你而言就是一门外语，你必须学习它的“语法和词汇表”，这就是本章和下一章要做的事情，但请不要忘记：



- 我们要学习的主要的是程序设计。
- 我们生产的是程序和系统。
- 程序设计语言（只）是工具。

记住这几点似乎极其困难，很多程序员对明显是属于编程语言语法和语义次要细节的内容表现出巨大的热情。特别是，很多人有这样一个错误观念：他们使用的第一种编程语言中的工作方法是做任何事的“不二法门”。请不要掉入这种陷阱。C++ 在很多方面是一种非常好的编程语言，但它并不完美，任何其他编程语言也都一样。



大多数程序设计概念是通用的，而很多这种概念被流行的程序设计语言所广泛支持。这意味着，我们在一门好的程序设计课中学到的基本思想和技术可以从一种编程语言延续到下一种语言。也就是说，它们可以（不同程度）方便地用于所有语言。而编程语言的技术细节则只限于特定的语言。幸运的是，编程语言不是凭空设计出来的，因此你在这里学到的很多内容都会在其他语言中找到明显的对应内容。特别是 C++ 与 C (第 27 章)、Java 和 C# 属于同一类语言，它们具有相当多的共性。

注意，当我们讨论语言技术问题时，我们故意使用非描述性的命名，如 `f`、`g`、`X` 和 `y`。我们这样做是为了强调这些例子的技术性质，保证这些例子很简短，以及尽力避免将语言技术细节和真正的程序设计方法混淆而令你困惑。当你看到非描述性的名字（例如那些永远不应该用在真实代码中的名字）时，请将注意力放在代码的语言技术层面上来。典型的语言技术实例仅仅包含用来展示语言规则的代码。如果你编译并运行这些代码，你会得到很多“变量未使用”的警告，而这样的技术性程序片段很少具有实用意义。

请注意我们在这两章介绍的内容不是 C++ 语法和语义的完整描述，甚至不是本书介绍的 C++ 功能的完整描述。ISO C++ 标准的篇幅超过 1300 页，充满晦涩的技术语言；而 Stroustrup 所著的《The C++ Programming Language》一书有 1300 多页，针对的是有经验的程序员。两者都包含 C++ 语言本身以及标准库。本书不会在完备性和综合性方面与它们一争高下，本书的优势是易懂，花在阅读上的时间有所值。

8.2 声明和定义

声明 (declaration) 语句将名字引入作用域 (8.4 节)，其作用是：

- 为命名实体 (如变量、函数) 指定一个类型。
- (可选) 进行初始化 (如为变量指定一个初始值，或为函数指定函数体)。

下面即为声明语句的例子：

```
int a = 7;           // int 型变量
const double cd = 8.7; // 双精度浮点常量
double sqrt(double); // 函数声明，参数为 double 型
                     // 返回值为 double 型
vector<Token> v;   // Token vector 的变量
```

C++ 程序中的名字都必须先声明后使用。考虑下面代码：

```
int main()
{
    cout << f(i) << '\n';
}
```

编译器最少会给出三个“未声明的标识符”的错误，因为在此程序片段中任何地方都没有 cout、f 和 i 的声明。我们可以通过包含头文件 std_lib_facilities.h 得到 cout 的声明：

```
#include "std_lib_facilities.h" // cout 声明在这里

int main()
{
    cout << f(i) << '\n';
}
```

现在，只剩两个“未定义”错误了。当你编写实用程序时，你会发现大多数声明都是在头文件中给出的。一般地，对于在“其他地方”定义的有用的功能，可以在头文件中为其定义接口。大致来说，一个声明定义了一些功能的使用方式，也就是，定义了函数、变量或类的接口。请注意声明的这种用途有一个明显的但易被忽视的优点：我们不必了解 cout 及其 << 操作符的定义的细节，我们只需用 #include 包含它们的声明即可。我们甚至无须真正了解它们的声明，从教材、手册、代码实例或其他资料中了解 cout 应如何使用就够了。编译器会读取头文件中声明，以便“理解”我们的程序。

现在，我们还需要声明 f 和 i，可以这样做：

```
#include "std_lib_facilities.h" // cout 声明在这里

int f(int);                  // 声明 f

int main()
{
    int i = 7;                // 声明 i
    cout << f(i) << '\n';
}
```

这段代码就可以编译通过了，因为每个名字都已经声明过了，但还会链接失败（2.4 节），因为我们还没有定义函数 f()；也就是说，我们还没有指出 f() 实际上应该做什么。

如果一个声明（还）给出了声明的实体的完整描述的话，我们称之为定义（definition）。

下面是一个定义的例子：

```
int a = 7;
vector<double> v;
double sqrt(double d) /* ... */
```

每个定义同时也是一个声明（根据我们对“定义”的定义），但某些声明不是定义。下面列出一些不是定义的声明，每个声明都需要在代码的其他位置给出对应的定义。

```
double sqrt(double); // 此处没有函数体
extern int a; // "extern" 加上没有初始化" 意味着 "未定义"
```

当我们对比定义和声明时，我们按惯例用“声明”表示“不是定义的声明”，即使这有点不那么严谨。

一个定义确切指明了一个名字代表什么。特别地，一个变量定义会为该变量分配存储空间。因此，你不能重复定义某个名字，例如：

```
double sqrt(double d) /* ... */ // 定义
double sqrt(double d) /* ... */ // 错误：重复定义

int a; // 定义
int a; // 错误：重复定义
```

与之相对，一个非定义声明仅仅告诉你如何使用一个名字，它只是一个接口，不会为变量分配存储空间或为函数指定函数体。因此，你可以声明一个名字任意多次，只要一致即可：

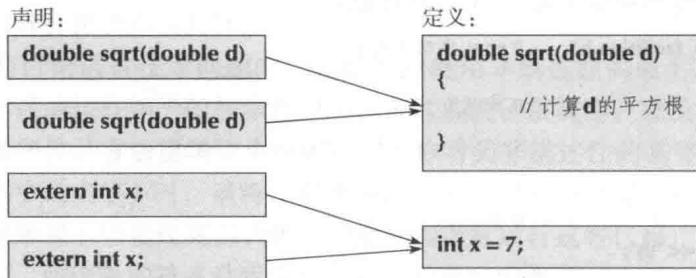
```
int x = 7; // 定义
extern int x; // 声明
extern int x; // 另一个声明

double sqrt(double); // 声明
double sqrt(double d) /* ... */ // 定义
double sqrt(double); // sqrt 的另一个声明
double sqrt(double); // sqrt 的再一个声明

int sqrt(double); // 错误：sqrt 不一致的声明
```

最后一个声明为什么是错误的？因为不允许这种情况发生：两个函数都叫 sqrt，接受一个同样是 double 类型的参数，但却返回不同类型的值（int 和 double）。

x 的第二个声明中使用的关键字 extern 表示此声明不是一个定义。这个关键字几乎没什么用处，我们建议你不要使用它，但是你会在别人的代码中看见它，特别是那些使用了非常多全局变量的代码（参见 8.4 节和 8.6.2 节）。



为什么 C++ 既提供声明功能，又提供定义功能呢？这两者间的区别反映出“如何使用一个实体（接口）”与“这个实体如何完成它应该做的事情（实现）”之间的根本区别。对于一个变量来说，其声明仅仅提供了类型，只有定义才能提供对象（存储空间）。对于一个函数来说，其声明也是只提供了类型（参数类型和返回类型），只有定义才提供函数体（可执行的语句）。注意，函数体是作为程序的一部分保存在内存中的，因此可以说函数和变量定义消耗了内存，而声明却没有。

声明和定义之间的区别使我们可以将一个程序分为很多部分，分开编译。声明功能使程序的每个部分都能保有程序其他部分的一个视图，而不必关心其他部分中的定义。所有声明（包括唯一的那个定义）必须一致，而整个程序中实体的命名也应一致。我们将在 8.3 节中进一步讨论这个问题。在此，我们只是提醒你回顾一下第 6 章中的表达式分析器：其中 `expression()` 调用 `term()`，`term()` 调用 `primary()`，而 `primary()` 又调用了 `expression()`。由于在 C++ 程序中每个名字都要先声明再使用，所以简单地定义这三个函数是行不通的：

```
double expression();      // 只是一个声明，而非定义
double primary()
{
    ...
    expression();
    ...
}
double term()
{
    ...
    primary();
    ...
}
double expression()
{
    ...
    term();
    ...
}
```

我们可以按任意顺序排列这四个函数，无论如何必然会在一个函数调用在其后定义的函数。这里，我们需要“前置声明”（forward declaration）。因此，我们在 `primary()` 的定义前声明 `expression()`，这样就一切顺利了。在实际编程中，这种循环调用非常常见。

为什么名字需要在使用前声明呢？为什么我们不能要求编译器通过读程序（就像我们做的那样）找出定义，来获得函数应该如何调用的信息呢？我们当然可以这样要求，但这会导致“有趣”的技术问题，所以我们决定不这样做。C++ 规范要求名字在使用前定义（类成员除外，参见 9.4.4 节）。毕竟，这种方式已经是一般写作（不是编写程序）的惯例了：当你阅读一本教材时，你当然希望作者在使用一个术语之前先定义它；否则，你不得不一直去猜测术语的含义或去查索引。“先声明后使用”的原则简化了阅读，无论是对人还是编译器。在一个程序中，“先声明后使用”之所以重要还有另外一个原因。在一个几千行（甚至几十万行）的程序中，很多我们希望调用的函数都是在“别处”定义的。而这个“别处”，我们往往不希望知道到底是哪。只需了解我们使用的实体的声明，把我们（还有编译器）从阅读大

量程序文本中解脱出来。

8.2.1 声明的类别

C++ 允许程序员定义很多类别的实体，我们比较关心的有：

- 变量。
- 常量。
- 函数（参见 8.5 节）。
- 名字空间（参见 8.7 节）。
- 类型（类和枚举，参见第 9 章）。
- 模板（参见第 14 章）。

8.2.2 变量和常量声明

一个变量或常量声明指定一个名字和一个类型，并可进行初始化。例如：

```
int a;                      // 不带初始化
double d = 7;                // 使用 = 语法进行初始化
vector<int> vi(10);         // 使用 () 语法进行初始化
vector<int> vi2 {1,2,3,4};    // 使用 {} 语法进行初始化
```

你可在 ISO C++ 标准中找到完整语法。

常量声明的语法与变量声明一样，差别在于类型之前多了一个关键字 `const`，而且必须进行初始化：

```
const int x = 7;              // 使用 = 语法进行初始化
const int x2 {9};              // 使用 {} 语法进行初始化
const int y;                  // 错误：未初始化
```

 必须进行初始化的原因是显然的：如果一个常量没有值的话，它何以为常量呢？对变量也进行初始化通常是个好主意，未初始化的变量常会导致隐蔽的错误。例如：

```
void f(int z)
{
    int x;                    // 未初始化变量
    // ... 此处不含对 x 的赋值语句 ...
    x = 7;                   // 对 x 赋值
    // ...
}
```

这段代码看起来再正常不过，但如果在第一个“...”处包含对 `x` 的使用又如何呢？例如：

```
void f(int z)
{
    int x;                    // 未初始化
    // ... 此处不含对 x 的赋值语句 ...
    if (z > x) {
        // ...
    }
    // ...
    x = 7;                   // 对 x 赋值
    // ...
}
```

因为 `x` 未初始化，所以执行 `z>x` 的结果是未定义的。在不同的机器平台上，比较操作 `z>x` 会给出不同的结果，甚至同一台机器上执行多次也会给出不同的结果。原则上，`z>x` 应导致程序因一个硬件错误而终止，但多数时候这不会发生，取而代之的是我们会得到一个不可预知的结果。

我们自然不会故意这么做，但我们可能犯错误，如果没有坚持初始化变量，上述情况就会发生。记住，很多“愚蠢的错误”（比如对于一个未初始化的变量，在对其赋值之前就使用它）都是在你很忙或疲倦的时候发生的。编译器会尽力给出警告，但对于复杂的代码（这类错误最可能发生的地方），编译器还无力捕捉所有这种错误。有的人不习惯初始化变量，这通常是因为他们学习程序设计所用的语言不允许或不鼓励一致的初始化；因此你会在别人的代码中看到这样的例子。请不要因为忘记初始化你自己定义的变量，而在你的程序中引入错误。

我们倾向于使用 `{} 初始化语法`。这是最一般的语法，而且它很明显地表明正在进行初始化。对于非常简单的初始化，有时候出于旧有习惯使用 `= 语法`；为了指定一个 `vector` 的元素个数，要使用 `() 语法`（见 12.4.4 节）。除此之外，我们最好都使用 `{} 语法` 来初始化。

8.2.3 默认初始化

你可能已经注意到了，我们通常不对 `string`、`vector` 等对象进行初始化。例如：

```
vector<string> v;
string s;
while (cin>>s) v.push_back(s);
```

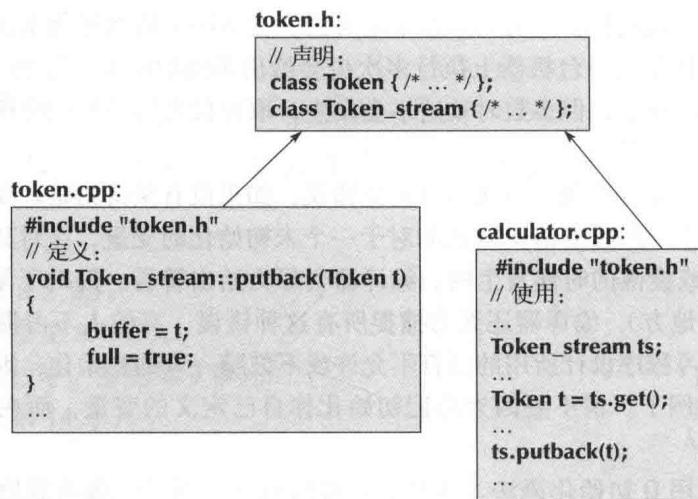
这并不是“变量必须先初始化再使用”这条规则的例外情况。之所以出现这种情况，是因为我们定义 `string` 类型和 `vector` 类型时定义了默认初始化机制，如果代码中不显式进行初始化，这两种对象就会被用一个默认值进行初始化。因此，上述代码执行到循环时，`v` 的值为空（不包含任何元素），`s` 的值为空串（`""`）。保证默认初始化的机制称为默认构造函数，参见 9.7.3 节。

不幸的是，C++ 不允许我们对内置类型设置默认初始化功能。全局变量会被默认初始化为 0，但你应该尽量少用全局变量。而最常使用的变量——局部变量和类成员——是不会被初始化的，除非你对其进行初始化（或提供一个默认构造函数）。我们已经提示过你了，你在实践中一定要注意！

8.3 头文件

我们如何管理声明和定义呢？这是一个大问题，因为声明和定义要一致，而实际程序可能会有数万个声明，甚至几十万个也不罕见。一般地，当我们编写程序时，我们使用的定义多数都不是我们自己写的。例如，`cout` 和 `sqrt()` 的实现就是别人在很多年前写的，我们只是使用它们。

在 C++ 中，对于“别处”定义的功能的声明，管理它们的关键是“头部”。本质上，一个头部（header）是一些声明的集合，一般定义于一个文件，因此也称为头文件（header file）。这样的头文件随后用 `#include` 包含在我们的源文件中。例如，我们可能决定改进计算器程序（第 6 章和第 7 章）的源码组织，将单词管理部分分隔出去。我们可以定义一个包含 `Token` 类和 `Token_stream` 类的声明的头文件 `token.h`：



`Token` 和 `Token_stream` 的声明在头文件 `token.h` 中，而它们的定义在 `token.cpp` 中。后缀 `.h` 通常用于 C++ 头文件，而 `.cpp` 后缀通常用于 C++ 源文件。实际上，C++ 语言并不关心文件后缀，但一些编译器和很多程序开发环境坚持这种命名习惯，因此你的源码组织也请遵循这一惯例。

原则上，`#include "file.h"` 只是简单地将 `file.h` 中的声明复制到你的文件中 `#include` 指令处。例如，我们可以写一个头文件 `f.h`：

```
// f.h
int f(int);
```

并将它包含于我们的源文件 `user.cpp` 中：

```
// user.cpp
#include "f.h"
int g(int i)
{
    return f(i);
}
```

当编译 `user.cpp` 时，编译器会执行包含操作，然后编译得到的如下程序

```
int f(int);
int g(int i)
{
    return f(i);
}
```

由于 `#include` 的处理逻辑上在编译器任何其他动作之前进行，因此被称为预处理 (preprocessing) (A.17 节)。

为了方便一致性检查，我们在使用声明的源文件和给出定义的源文件中都包含头文件。这样，编译器就能尽可能快地捕获错误。例如，假定实现 `Token_stream::putback()` 的程序员犯了如下错误：

```
Token Token_stream::putback(Token t)
{
    buffer.push_back(t);
    return t;
}
```

这段代码看起来没有问题（虽然它存在错误），幸运的是，编译器可以发现这个错误，因为它看到了（包含进来的）`Token_stream::putback()` 的声明。将之与这里的定义比较后，编译器发现 `putback()` 不应该返回一个 `Token`，另外 `buffer` 是一个 `Token` 而不是一个 `vector<Token>`，因此我们不能在其上使用 `push_back()`。这个错误产生的原因是，我们在原有的代码上继续工作，试图改进它，但进行的修改没有保证整个程序的一致性。

类似地，考虑下面代码中的错误：

```
Token t = ts.get();           // 错误：没有成员 gett
...
ts.putback();                // 错误：缺少参数
```

编译器会立即报告错误，因为头文件 `token.h` 给出了一致性检查所需的所有信息。

头文件 `std_lib_facilities.h` 包含了我们所使用的标准库中的功能的声明，如 `cout`、`vector` 和 `sqrt()`，以及一些不在标准库中的简单工具函数的声明，如 `error()`。17.8 节中我们会说明如何直接使用标准库头文件。

一个头文件通常会被包含在很多源文件中，这意味着头文件只能包含那些可以在多个文件中重复多次的声明（如函数声明、类定义和数值常量的定义）。

8.4 作用域

作用域（scope）是一个程序文本区域。每个名字都定义在一个作用域中，在声明点到作用域结束的区间内有效。例如：

```
void f()
{
    g();          // 错误：g() 不在作用域里
}

void g()
{
    f();          // 正确：f() 在作用域里
}

void h()
{
    int x = y;    // 错误：y 不在作用域里
    int y = x;    // 正确：x 在作用域里
    g();          // 正确：g() 在作用域里
}
```

名字在其声明的定义域嵌套的定义域中也有效。例如，上面代码中对 `f()` 的调用在 `g()` 的作用域中，此作用域嵌套于全局作用域。全局作用域不在任何其他作用域内。名字必须先声明后使用的规则在这里还是适用的，因此 `f()` 不能调用 `g()`。

C++ 支持多种类型的作用域，帮助我们控制变量在哪里可用：

- 全局作用域（global scope）：在任何其他作用域之外的程序区域。
- 名字空间作用域（namespace scope）：一个名字空间作用域嵌套于全局作用域或另一个名字空间作用域中，参见 8.7 节。
- 类作用域（class scope）：一个类内的程序区域，参见 9.2 节。
- 局部作用域（local scope）：位于 `{ ... }` 大括号之间或函数参数列表中的程序区域。
- 语句作用域（statement scope）：例如，`for` 语句内的程序区域。

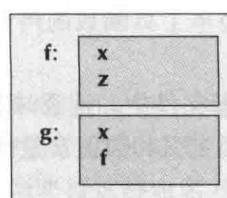
作用域的主要作用是保持名字的局部性，使之不影响声明于其他地方的名字。例如：

```
void f(int x)          // f 是全局的, x 在 f 的局部作用域里
{
    int z = x+7;      // z 是局部的
}

int g(int x)          // g 是全局的, x 在 g 的局部作用域里
{
    int f = x+2;      // f 是局部的
    return 2*f;
}
```

下图描述了上面代码中的作用域信息：

全局作用域：



这段代码中，`f()` 里的 `x` 与 `g()` 里的 `x` 是不一样的。它们不会冲突，因为不在同一个作用域中：`f()` 里的 `x` 在 `f` 的局部作用域中，而 `g()` 里的 `x` 在 `g` 的局部作用域中。位于同一个作用域中，不能共存的两个声明被称为冲突（clash）。类似地，在 `g()` 中定义、使用的 `f`（显然）不是全局函数 `f()`。

下面代码中局部作用域的使用与上例类似，但这段代码更接近实用：

```
int max(int a, int b)      // max 是全局的, a 和 b 是局部的
{
    return (a>=b) ? a : b;

int abs(int a)            // 不是 max() 中的 a
{
    return (a<0) ? -a : a;
}
```

你在标准库中会发现 `max()` 和 `abs()`，因此你不必自己编写这两个函数。`?:` 结构称作算术 if (arithmetic if) 或条件表达式 (conditional expression)。当 `a>=b` 时，`(a>=b)?a:b` 的值为 `a`，否则为 `b`。条件表达式可以帮助我们避免下面这种冗长的代码：

```
int max(int a, int b)      // max 是全局的, a 和 b 是局部的
{
    int m;                  // m 是局部的
    if (a>=b)
        m = a;
    else
        m = b;
    return m;
}
```

因此，除了很明显的全局作用域外，其他作用域都使名字保持局部性。在很多场合，局部性是一种很好的性质，因此你应该尽量保持名字的局部性。当我在函数、类和名字空间内部

声明变量和函数时，它们不会影响你的变量和函数。记住，实际程序中有成千上万的命名实体，为了使这种程序易于管理，多数名字都应该是局部的。

下面是一个较大的实例，说明了名字是如何在语句和语句块（包括函数体）末尾离开作用域的：

```
// 此处没有 r、i、v
class My_vector {
    vector<int> v; // v 在类作用域里
public:
    int largest()
    {
        int r = 0; // r 是局部的（最小的非负整数）
        for (int i = 0; i < v.size(); ++i)
            r = max(r, abs(v[i])); // i 在 for 语句的作用域里
        // no I here
        return r;
    }
    // no r here
};

// no v here

int x; // 全局变量——应尽可能避免使用
int y;

int f()
{
    int x; // 局部变量，使得全局的 x 不可见
    x = 7; // 局部变量 x
    {
        int x = y; // 局部变量 x，由全局变量 y 初始化，使前一个局部变量 x 不可见
        ++x; // 上一行的 x
    }
    ++x; // f() 中第一行的 x
    return x;
}
```

只要可能，你应该避免这种复杂的嵌套和隐藏。记住：“保持简单性！”

一个名字的作用域越大，名字就应该越长、越有描述性：将全局变量命名为 `x`、`y` 和 `f` 是灾难性的。你在程序中应尽量少用全局变量，一个主要原因是很难知道哪个函数会修改它们。在大的程序中，基本不可能知道哪个函数修改了一个全局变量。想象你正在调试一个程序，而你发现一个全局变量的值与预期不符。那么是谁赋予它这个值？为什么这样赋值？是哪个函数干的？你如何能知道这些信息？赋予此变量这个错误值的函数可能在你从未见过的一个源文件中！如果非有不可的话，一个好的程序也应该只有非常少（比如说一个或两个）的全局变量。例如，我们在第 6 章和第 7 章给出的计算器程序只有两个全局变量：单词流 `ts` 和符号表 `names`。

注意，很多 C++ 语法结构定义了嵌入的作用域：

- 类中的函数：成员函数（参见 9.4.2 节）。

```
class C {
public:
    void f();
    void g() // 在类中定义的成员函数
{
```

```

        //...
    }
    //...
};

void C::f()      //在类外定义的成员函数
{
    //...
}

```

这是一种最常见和最有用的情况。

- 类中的类：成员类（也称作嵌入类）。

```

class C {
    public:
        struct M {
            //...
        };
        //...
};

```

这只在复杂类中才有用，记住理想情况是保持类简短、简单。

- 函数中的类：局部类。

```

void f()
{
    class L {
        //...
    };
    //...
}

```



应避免这种代码，如果你觉得需要一个局部类，那么你的函数可能太长了。

- 函数中的函数：局部函数（也称作嵌套函数）。

```

void f()
{
    void g()      //非法
    {
        //...
    }
    //...
}

```

这在 C++ 中是不合法的，不要写这种代码，编译器会拒绝它。

- 函数或其他块中的块：嵌套块。

```

void f(int x, int y)
{
    if (x>y) {
        //...
    }
    else {
        //...
    }
    //...
}

```

嵌套块是避免不了的，但要对复杂的嵌套保持警惕：它很容易隐藏错误。

C++ 还提供了一种语言特性：名字空间，专门用于表达作用域，参见 8.7 节。

注意，我们使用一致的缩进格式来表明嵌套，如果不这样，嵌套的结构就会很难读。例如：

```
// 危险、丑陋的代码
struct X {
    void f(int x) {
        struct Y {
            int f() { return 1; } int m; };
            int m;
            m=x; Y m2;
            return f(m2.f()); }
            int m; void g(int m) {
                if (m) f(m+2); else {
                    g(m+2); }}
                    X() {} void m3() {
                    }

void main() {
    X a; a.f(2);}
    };
```

难读的代码往往隐藏着错误。当你使用某种 IDE 时，IDE 会尽力自动地（根据某种“恰当的”规则）将你的代码整理成恰当的缩进格式。也有一种“代码美化器”，可以重整源代码文件的格式（通常可以允许你自己选择格式）。但是，令你的代码可读的责任最终还是在你自己。

8.5 函数调用和返回

函数为我们提供了表示操作和计算的途径。当我们完成某些工作，而这些工作值得我们为它起一个名字的话，我们就可以编写一个函数。C++ 语言为我们提供了运算符（如 + 和 *），我们可以在表达式中用运算符从运算对象产生出新值。C++ 还提供了语句（如 for 和 if），我们可以用之控制程序执行的顺序。为了组织由这些原语构成的代码，我们需要使用函数。

为了完成自身的工作，函数通常需要参数，很多函数还返回一个结果。本节关注参数如何指定和传递。

8.5.1 声明参数和返回类型

函数是 C++ 中我们用来命名和表示计算和操作的语法结构。一个函数声明由一个返回值后跟函数名，再接一个形式参数（formal argument，简称形参）列表构成。例如：

```
double fct(int a, double d);           // 声明 fct (无函数体)
double fct(int a, double d) { return a*d; } // 定义 fct
```

一个函数定义包含函数体（调用此函数时应执行的语句），而一个非定义的函数声明则只接一个分号。形参通常称为参数（parameter）。如果你不希望一个函数接受参数，则可省略形参。例如：

```
int current_power();           // current_power 没有参数
```

如果你不希望函数返回一个结果，可将其返回类型设置为 void。例如：

```
void increase_power(int level); // increase_power 不返回一个值
```

这里, **void** 的意思是“不返回一个值”或“什么也不返回”。

在函数声明和定义中, 你可以为参数命名也可以不命名, 完全取决于你的需要。例如:

```
// 在 vs 中查找 s
// vs[hint] 可能是一个好的开始查找位置
// 返回匹配的索引值。-1 表示“没找到”
int my_find(vector<string> vs, string s, int hint); // 命名参数

int my_find(vector<string>, string, int); // 不命名参数
```

在函数声明中, 形参的名字不是必需的, 只是对于编写好的注释很有益处。以编译器的观点, 第二个 **my_find()** 声明与第一个一样好: 它包含所有调用 **my_find()** 所需的信息。

通常, 我们会命名函数定义中的所有形参, 例如:

```
int my_find(vector<string> vs, string s, int hint)
// 在 vs 中从 hint 位置处开始查找 s
{
    if (hint<0 || vs.size()<=hint) hint = 0;
    for (int i = hint; i<vs.size(); ++i) // 从 hint 位置处开始查找
        if (vs[i]==s) return i;
    if (0<hint) { // 如果没找到 s, 则搜索 hint 位置之前的值
        for (int i = 0; i<hint; ++i)
            if (vs[i]==s) return i;
    }
    return -1;
}
```

参数 **hint** 使代码复杂了许多, 但它的使用是基于这样一个假设: **my_find()** 的使用者粗略知道如何在一个 **vector** 中找到一个字符串, 所以使用 **hint** 可以导致好的效果。但是, 设想我们已经使用了 **my_find()** 一段时间, 发现调用者很少能用好 **hint**, 因此它实际上降低了性能。现在我们不再需要 **hint** 了, 但是已有大量“外部”代码调用 **my_find()** 时使用了参数 **hint**。我们不希望重写这些代码(或者因为是他人所写代码无法修改), 因此我们不想更改 **my_find()** 的声明。替代方法是, 我们不再使用最后一个参数(但在声明中保留它)。由于不再使用, 所以可以不为其命名:

```
int my_find(vector<string> vs, string s, int) // 不使用第 3 个参数
{
    for (int i = 0; i<vs.size(); ++i)
        if (vs[i]==s) return i;
    return -1;
}
```

你可在 ISO C++ 标准中找到函数定义的完整语法。

8.5.2 返回一个值

我们可以用 **return** 语句从函数返回一个值:

```
T f() // f() 返回一个 T 类型的值
{
    V v;
    // ...
    return v;
}
T x = f();
```

这段代码中的返回值恰好就是我们用一个类型为 V 的值初始化一个类型为 T 的变量所得到的值：

```
V v;
// ...
T t(v); // 用 v 初始化 t
```

也就是说，返回值可以看作初始化的另一种形式。

如果函数声明中指定要返回值，则函数体内必须通过 `return` 返回一个值。否则，就会导致错误“直至函数末尾未返回值”：

```
double my_abs(int x) // 警告：有问题代码
{
    if (x < 0)
        return -x;
    else if (x > 0)
        return x;
} // 错误：当 x 为 0 时，没有返回值
```

实际上，编译器可能不会注意到我们“忘记了”`x==0`的情形。原则上它可以注意到，但很少有编译器如此聪明。对于复杂的函数，编译器完全可能无法知道你是否返回了一个值，因此编程中要小心。这里，“小心”的意思是，要切实保证对于函数的每种执行路径都有一条 `return` 语句或者一个 `error()`。

由于历史原因，`main()` 是一个特例。执行到 `main()` 的末尾而未返回值，等价于返回 0，意思是“成功完成”程序。

在一个不返回值的函数中，我们可以调用无值的 `return` 语句从函数返回调用者。例如：

```
void print_until_s(vector<string> v, string quit)
{
    for(int s : v) {
        if (s==quit) return;
        cout << s << '\n';
    }
}
```

如你所见，在一个 `void` 函数中直至末尾未返回值是合法的，这等价于一个无值的返回 `return;`。

8.5.3 传值

向函数传递参数最简单的方式是，将参数的值拷贝一份交给函数。一个函数 `f()` 的参数实际上是 `f()` 中的局部变量，每次 `f()` 被调用时都会初始化。例如：

```
// 传值方式（将待传值拷贝一份，交给函数）
int f(int x)
{
    x = x+1; // 对局部变量 x 赋值
    return x;
}

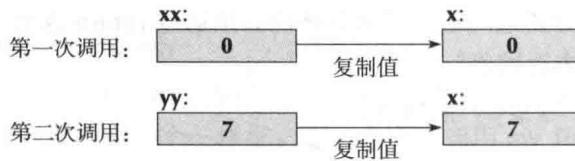
int main()
{
    int xx = 0;
    cout << f(xx) << '\n'; // 输出：1
    cout << xx << '\n'; // 输出：0；f() 不改变 xx
```

```

int yy = 7;
cout << f(yy) << '\n';           // 输出: 8
cout << yy << '\n';           // 输出: 7; f() 不改变 yy
}

```

由于传递的是拷贝，因此 f() 中的 `x=x+1` 不会改变两次调用时传递的变量 `xx` 和 `yy` 的值。下图可以说明按值参数传递的机制：



传值方式非常直接，其代价就是复制值的开销。

8.5.4 传常量引用

当传递占用存储空间小的值，如一个整数、一个双精度数或者一个单词（6.3.2节）时，传值方式简单、直接、高效。但对于占用存储空间大的值，如一个图像（通常有几百万位大小）、一个大表（比如说几千个整数）或一个长字符串（比如说几百个字符）时，又如何呢？这种情况下，拷贝的代价就会非常高。我们不必为拷贝代价所困扰，但做不必要的工作就可能会很麻烦了，这意味着我们不能直接表达我们想要什么。例如，我们可能会编写下面这个函数来打印一个浮点数 vector：

```

void print(vector<double> v)           // 传值方式：是否合适？
{
    cout << "{ ";
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i];
        if (i != v.size() - 1) cout << ", ";
    }
    cout << " }\n";
}

```

这个 `print()` 函数适用于所有规模的 `vector`，例如：

```

void f(int x)
{
    vector<double> vd1(10);      // 小 vector
    vector<double> vd2(1000000); // 大 vector
    vector<double> vd3(x);       // 未知大小的 vector
    // ... 为 vd1、vd2、vd3 赋值 ...
    print(vd1);
    print(vd2);
    print(vd3);
}

```

这段代码可以得到我们想要的结果，但首次调用 `print()` 需要拷贝 10 个双精度数（大概 80 个字节），第二次调用需要拷贝一百万个双精度数（大概 8 兆字节），而第三次调用需要拷贝多少字节我们不知道。在此，我们必须问自己一个问题：“为什么我们要拷贝全部数据？”我们只不过是想打印 `vector`，而不是拷贝它们的元素。显然，必须要有一种方法，能使我们向函数传递一个变量，而不拷贝其值。一个类似的例子是，如果你被分派了一个任务——为

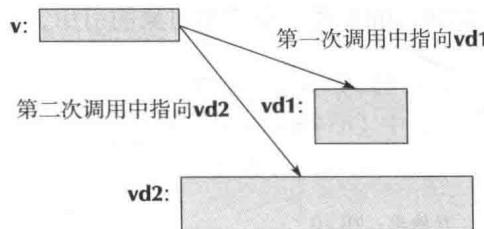
图书馆中的书籍建一个目录，馆长不会给你送去图书馆大楼和其内所有内容的一份拷贝，而只是把图书馆的地址发送给你，这样你就能到图书馆去浏览馆藏书籍。因此，我们需要某种方法，能将要打印的 `vector` 的“地址”而不是其拷贝传送给 `print()` 函数。这样的“地址”被称为引用 (reference)，其使用方法如下：

```
void print(const vector<double>& v) // 传常量引用方式
{
    cout << "{ ";
    for (int i = 0; i < v.size(); ++i) {
        cout << v[i];
        if (i != v.size() - 1) cout << ", ";
    }
    cout << " }\n";
}
```

符号 `&` 表示“引用”，而此处的 `const` 用来阻止 `print()` 无意中修改其参数。除了参数声明进行了修改外，代码所有其他部分都与传值方式的版本完全一样；唯一的改变在于 `print()` 现在是通过引用“提取到”参数的值，而非拷贝操作。注意短语“提取” (refer back)，这种参数之所以被称为引用，是因为它们“指向”定义于别处的对象 (它们并不是对象本身)。我们可以像以前一样调用新版本的 `print()`：

```
void f(int x)
{
    vector<double> vd1(10);           // 小 vector
    vector<double> vd2(1000000);       // 大 vector
    vector<double> vd3(x);           // 未知大小的 vector
    // ... 为 vd1、vd2、vd3 赋值 ...
    print(vd1);
    print(vd2);
    print(vd3);
}
```

传常量引用方式可图示如下：



常量 (`const`) 引用的一个非常有用特性是，我们不可能意外地修改传来的对象。例如，如果我们犯了一个愚蠢的错误，试图在 `print()` 中修改 `vector` 元素，编译器就会发现这个错误：

```
void print(const vector<double>& v) // 传常量引用方式
{
    // ...
    v[i] = 7;                         // 错误：v 是一个常量（不可修改）
    // ...
}
```

传常量引用参数是一种有用的、常用的机制。请再次考虑 `my_find()` 函数 (8.5.1 节)，

它在一个字符串 vector 中搜索一个字符串，传值参数会导致不必要的拷贝代价：

```
int my_find(vector<string> vs, string s); // 传值方式：需要拷贝
```

如果 vector 中包含数千个字符串，即便在一台非常快的计算机上，你也能观察到拷贝所花费的时间。因此，我们可以通过将 my_find() 的参数改为传常量引用方式来改进它：

// 传常量引用方式：不需要拷贝，只读

```
int my_find(const vector<string>& vs, const string& s);
```

8.5.5 传引用

但是，如果我们确实希望函数修改其参数，又该怎么办呢？有时，我们有充足的理由需要这么做。例如，我们可能需要一个 init() 函数为 vector 元素赋值：

```
void init(vector<double>& v) // 传引用
{
    for (int i = 0; i < v.size(); ++i) v[i] = i;
}

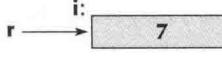
void g(int x)
{
    vector<double> vd1(10); // 小 vector
    vector<double> vd2(1000000); // 大 vector
    vector<double> vd3(x); // 未知大小的 vector

    init(vd1);
    init(vd2);
    init(vd3);
}
```

这里，我们希望 init() 函数修改参数 vector，因此我们没有使用传值参数（拷贝参数值），也没有使用传常量引用参数（不允许修改参数），只是将实际参数的“简单引用”传递给形参。

让我们从更技术化的角度来探讨一下引用。引用是这样一种语法机制，它允许用户为一个对象声明一个新的名字。例如，int& 是一个整型对象的引用，因此，我们可写出如下代码

```
int i = 7;
int& r = i; // r 是 i 的引用
r = 9; // i 变为 9
i = 10;
cout << r << ' ' << i << '\n'; // 输出：10 10
```



也就是说，任何对 r 的使用实际上使用的是 i。

引用的一个用途是作为简写形式。例如，我们可能用到如下二维 vector

```
vector<vector<double>> v; // double 型 vector 的 vector
```

我们需要多次使用某个 vector 元素 v[f(x)][g(x)]。v[f(x)][g(x)] 是一个复杂的表达式，我们当然不愿意反复输入它。如果我们只是需要这个元素的值，那么可以声明下面这个变量

```
double val = v[f(x)][g(y)]; // val 是 v[f(x)][g(x)] 的值
```

然后多次使用 val 即可。但如果我们既要从 v[f(x)][g(x)] 读取值，又要向它写入值呢？这时，引用就派上用场了：

```
double& var = v[f(x)][g(y)]; // var 是 v[f(x)][g(x)] 的引用
```

现在，通过 var，我们既可以从 v[f(x)][g(x)] 读，也可以向它写。例如：

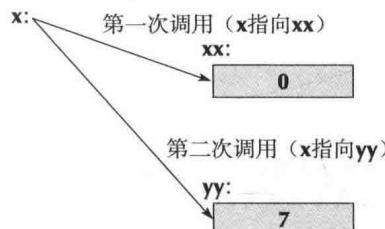
```
var = var/2+sqrt(var);
```

引用的这一重要特性，即可以方便地作为某个对象的简写形式的特性，是引用能作为一种有用的参数传递方式的原因。例如：

```
// 传引用方式（函数可访问传入变量本身）
int f(int& x)
{
    x = x+1;
    return x;
}
int main()
{
    int xx = 0;
    cout << f(xx) << '\n';           // 输出：1
    cout << xx << '\n';           // 输出：1；f() 改变了 xx 的值

    int yy = 7;
    cout << f(yy) << '\n';           // 输出：8
    cout << yy << '\n';           // 输出：8；f() 改变了 yy 的值
}
```

下图说明了上例中传引用方式参数传递的原理：



请将此例与 8.5.3 中的类似实例进行比较。

传引用参数显然是一种非常强大的机制：在函数中我们可以直接操作任何以引用方式传递来的对象。例如，交换两个值是很多算法（例如排序）中非常重要的操作。利用引用，我们可以编写下面这样一个交换两个浮点数的函数：

```
void swap(double& d1, double& d2)
{
    double temp = d1;           // 拷贝 d1 的值到 temp
    d1 = d2;                   // 拷贝 d2 的值到 d1
    d2 = temp;                 // 拷贝 d1 的旧值到 d2
}

int main()
{
    double x = 1;
    double y = 2;
    cout << "x == " << x << " y == " << y << '\n'; // 输出：x==1 y==2
    swap(x,y);
    cout << "x == " << x << " y == " << y << '\n'; // 输出：x==2 y==1
}
```

标准库提供了一个 `swap()` 函数，可以用来交换任何类型的值，只要该类型支持拷贝操作即可。因此，你不需要为每个类型编写自己的 `swap()` 函数。

8.5.6 传值与传引用的对比

如何在传值方式、传引用方式和传常量引用方式间进行选择呢？我们先来看第一个例子：

```
void f(int a, int& r, const int& cr)
{
    ++a;      // 改变局部变量 a
    ++r;      // 改变 r 指向的对象
    ++cr;     // 错误：cr 是常量引用
}
```

如果你希望改变被传递的对象的值，你应该使用非常量的引用：传值方式传来的是对象的拷贝，而传常量引用方式不允许你修改对象的值。你可以试试下面的程序，观察三种参数传递方式的效果：

```
void g(int a, int& r, const int& cr)
{
    ++a;      // 改变局部变量 a
    ++r;      // 改变 r 指向的对象
    int x = cr; // 读入 cr 指向的对象
}

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;

    g(x,y,z); // x==0; y==1; z==0
    g(1,2,3); // 错误：引用参数 r 需要指向一个变量
    g(1,y,3); // 正确：因为 cr 是常量引用，可以传递一个字面常量
}
```

因此，如果想改变通过引用方式传递过来的对象的值，你必须传递一个对象。从技术上讲，整型字面常量 2 只是一个值（右值，rvalue），而不是一个能保存值的对象。而这里函数 `g()` 的参数 `r` 需要的是一个左值（lvalue），也就是说，可以出现在赋值号左边的内容。

注意，常量引用不需要一个左值，它可以像初始化和传值方式一样进行转换。在上面代码中，当进行最后一次调用 `g(1, y, 3)` 时发生了什么呢？情况是这样的，编译器为函数 `g()` 的参数 `cr` 分配了一个整型变量，令 `cr` 指向它：

```
g(1,y,3); // 意味着：int _compiler_generated = 3; g(1,y,_compiler_generated)
```

这种编译器生成的对象称为临时对象（temporary object）。

我们的基本原则是：

- 1. 使用传值方式传递非常小的对象。
- 2. 使用传常量引用方式传递你不需修改的大对象。
- 3. 让函数返回一个值，而不是修改通过引用参数传递来的对象。
- 4. 只有迫不得已时才使用传引用方式。

这些原则会帮我们写出最简单、最不易出错而且最高效的代码。“非常小”的意思是一个或两个整型数，一个或两个双精度数，或者和它们差不多大小的对象。如果我们发现一个参数

是以非常量引用方式传递的，我们必须假设被调用的函数会修改这个参数。

第三条规则表达的是，当你想用函数改变一个变量的值时，实际上你还有另一种选择。考虑如下代码：

```
int incr1(int a) { return a+1; }      // 返回新的值作为结果
void incr2(int& a) { ++a; }          // 通过传引用修改对象

int x = 7;
x = incr1(x);                      // 意义非常明显
incr2(x);                          // 相当晦涩难懂
```

那么我们为什么还需要非常量引用传递方式呢？因为有时候这种参数传递方式是必要的。 

- 用于操作容器（比如 `vector`）和其他大的对象。
- 用于改变多个对象的函数（函数只能有一个返回值）。

例如：

```
void larger(vector<int>& v1, vector<int>& v2)
{
    // 将 v1 和 v2 中对应元素的较大者存储在 v1 中
    // 类似地，将 v1 和 v2 中对应元素的较小者存储在 v2 中
    if (v1.size()!=v2.size()) error("larger(): different sizes");
    for (int i=0; i<v1.size(); ++i)
        if (v1[i]<v2[i])
            swap(v1[i],v2[i]);
}

void f()
{
    vector<int> vx;
    vector<int> vy;
    // 从输入中读取 vx 和 vy
    larger(vx,vy);
    ...
}
```

对于 `larger()` 这样的函数来说，使用传引用参数是唯一合理的选择。

通常最好避免让函数修改多个对象。理论上，总会有替代方法，比如返回一个包含多个值的类对象。但是，已有大量程序使用了修改一个或多个参数的函数，因此你很可能遇到这类程序。例如，在 Fortran 语言（大约 50 年中一直是用于数值计算的主要编程语言）中，所有参数都是以引用方式传递的。很多数值计算程序直接借鉴了已有的 Fortran 程序，调用了 Fortran 函数。这些代码通常使用传引用方式和传常量引用方式。

如果我们使用引用只是想避免拷贝操作，那可以使用常量引用。这样，当我们看到一个非常量引用参数时，就可以假定这个函数改变了参数的值；也就是说，当看到一个非常量引用的参数传递时，我们假定这个函数不仅是可以修改参数的值，而是确实这么做了，因此我们必须小心检查对函数的调用，确定它按我们所期待的那样工作。 

8.5.7 参数检查和转换

参数传递过程就是用函数调用中指定的实际参数（actual argument）初始化函数的形式参数的过程，考虑如下代码：

```
void f(T x);
f(y);
T x=y;                         // 用 y 初始化 x (见 8.2.2 节)
```

只要初始化语句 `T x=y;` 合法，函数调用 `f(x)` 就是合法的，当其合法时，两个 `x`（初始化的变量和函数的参数）会获得相同的值。例如：

```
void f(double x);
void g(int y)
{
    f(y);
    double x = y; // 用 y 初始化 x (见 8.2.2 节)
}
```

注意，用 `y` 初始化 `x` 时，我们必须将一个整数转换为一个双精度数。在调用函数 `f()` 时，会进行同样的操作。`f()` 收到的双精度值与变量 `x` 中保存的值是一样的。

 类型转换一般情况是很有用的，但偶尔会带来奇怪的结果（参见 3.9.2 节）。因此，我们对类型转换必须小心。例如，如果一个函数要求一个整数，那么向它传递一个双精度参数就不是一个好主意：

```
void ff(int x);

void gg(double y)
{
    ff(y); // 怎样知道操作是否合理?
    int x = y; // 怎样知道操作是否合理?

}
```

如果你确实是想将一个双精度值截取为一个整数，请使用显式类型转换：

```
void ggg(double x)
{
    int x1 = x; // 截断 x
    int x2 = int(x);
    int x3 = static_cast<int>(x); // 非常显式的转换 (12.8 节)

    ff(x1);
    ff(x2);
    ff(x3);

    ff(x); // 截断 x
    ff(int(x));
    ff(static_cast<int>(x)); // 非常显式的转换 (12.8 节)
}
```

使用显式类型转换的代码，其他程序员容易从中看出你的思路。

8.5.8 实现函数调用

当一个函数被调用时，计算机实际上做了什么呢？第 6 章和第 7 章中的函数 `expression()`、`term()` 和 `primary()` 可以很好地说明这一问题，除了一个细节：这些函数都不接受参数，因此我们无法用它们解释参数是如何传递的。但是，请等一等！这些函数必然是获取一些输入的，否则它们不可能做任何有用的事情。实际上它们接受了一个隐含的参数：它们使用了一个称为 `ts` 的 `Token_stream` 对象来获得输入，而 `ts` 是一个全局变量。这有点偷偷摸摸的。我们可以改进这些函数，让它们接受一个 `Token_stream&` 类型的参数。因此本节中这几个函数都被增加了一个 `Token_stream&` 参数，而所有与函数调用实现不相关的内容都被去掉了。

首先，函数 `expression()` 非常简单，它有一个参数 (`ts`) 和两个局部变量 (`left` 和 `t`)：

```
double expression(Token_stream& ts)
{
    double left = term(ts);
    Token t = ts.get();
    // ...
}
```

第二，函数 `term()` 与 `expression()` 非常类似，只是多了一个额外的局部变量 (`d`)，用来保存除法运算的除数。

```
double term(Token_stream& ts)
{
    double left = primary(ts);
    Token t = ts.get();
    // ...
    case '/':
    {
        double d = primary(ts);
        // ...
    }
    // ...
}
```

第三，函数 `primary()` 与 `term()` 很类似，只是多了一个局部变量 `left`:

```
double primary(Token_stream& ts)
{
    Token t = ts.get();
    switch (t.kind) {
        case '(':
            {   double d = expression(ts);
                // ...
            }
            // ...
    }
}
```

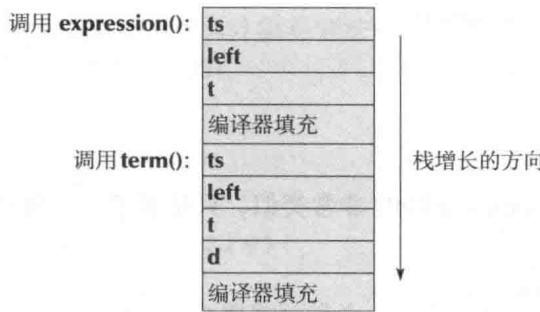
现在这些函数已经不再使用任何“偷偷摸摸的全局变量”了，用来说明函数调用机制已经非常理想了：它们都有一个参数，都有局部变量，而且它们相互调用。你可能希望有机会重新回顾一下完整的 `expression()`、`term()` 和 `primary()` 是什么样，但与函数调用相关的特性这里都已经给出了。

当一个函数被调用时，编译器分配一个数据结构，保存所有参数和局部变量的拷贝。例如，当 `expression()` 第一次被调用时，编译器会创建如下数据结构：

调用 <code>expression()</code> :	<table border="1"> <tr><td>ts</td></tr> <tr><td>left</td></tr> <tr><td>t</td></tr> <tr><td>编译器填充</td></tr> </table>	ts	left	t	编译器填充
ts					
left					
t					
编译器填充					

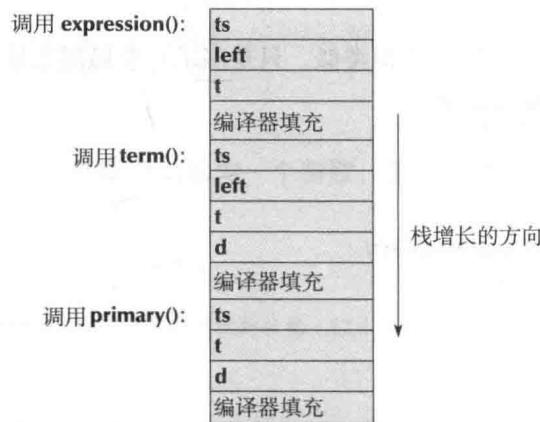
“编译器填充”部分，不同编译器填入的内容不同，但基本上是函数返回到调用者以及返回一个值给调用者所需的信息。这样的数据结构称为函数活动记录（function activation record），每个函数的活动记录都有自己特有的详细布局。注意，从编译器的角度，一个参数只是另一个局部变量而已。

到目前为止，一切都很好，现在 `expression()` 调用 `term()`，编译器会为 `term()` 的这次调用创建相应的活动记录：

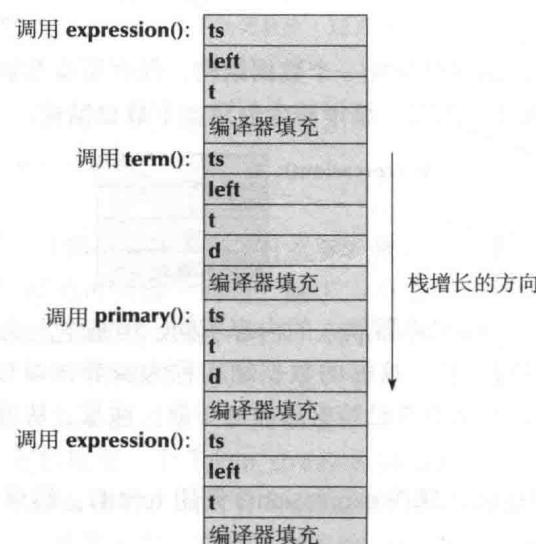


我们注意到 `term()` 需要保存一个额外的变量 `d`，因此在调用中编译器为它分配了存储空间，即使程序中可能永远也不使用它。这没有问题。对于合理的函数（比如本书中我们直接或间接使用的所有函数）来说，创建一个函数活动记录的运行时代价不依赖于它有多大。局部变量 `d` 只有当我们执行 case '`l`' 时才会被初始化。

现在 `term()` 调用 `primary()`，编译器会创建如下活动记录：

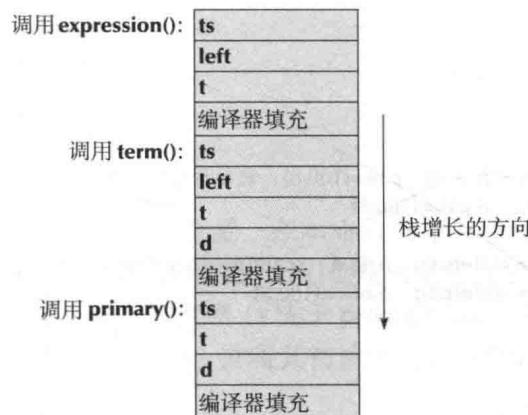


这么叙述下去看来有些啰嗦了，但现在 `primary()` 调用 `expression()`：

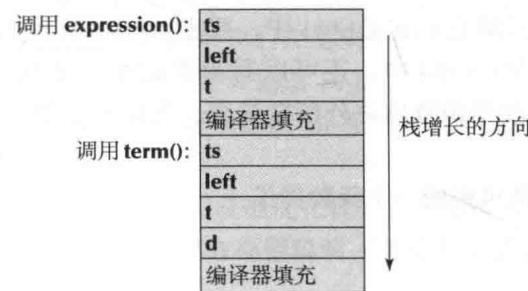


编译器为 `expression()` 的这次调用创建了它自己的活动记录，与第一次 `expression()` 调用的活动记录是不同的。这样 `left` 和 `t` 在两次调用中是不同的，这是一种很好的处理方式，否则我们将处于糟糕的境地。一个函数如果直接或间接（本例）调用自身的话，我们称之为递归（recursive）函数。如你所见，正是因为有了上述函数调用和返回的实现技术，递归函数才得以成立（反之亦然）。

因此，每当我们调用函数时，活动记录栈（stack of activation record）——通常就称为栈（stack）生长出一个记录。反过来，当函数返回时，其记录就不再有用。例如，当最后一个 `expression()` 调用返回 `primary()` 时，栈会复原为下图：



当 `primary()` 返回 `term()` 时，栈回到



依此类推。这里使用的栈也称为调用栈（call stack），是一种只在一端增长和收缩的数据结构，其增长和收缩的规则是先进先出。

请记住不同 C++ 编译器实现和使用调用栈的细节是不同的，但基本的原理就大致如上文所述。为了使用函数，你需要知道函数调用是如何实现的吗？当然不需要，你在前面学习的使用函数的知识已经足够多、足够好了，学习本小节关于函数调用实现方面的内容不是必需的，但很多程序员还是想知道这方面的知识，而且很多程序员使用诸如“活动记录”、“调用栈”之类的术语，所以了解一下这方面的内容还是有好处的。

8.5.9 `constexpr` 函数

一个函数代表一种计算。有时我们希望在编译时进行某种计算，通常这么做的目的是为了避免在运行时计算同样的内容上百万次。使用函数可使计算更易理解，很自然地，有时

我们希望在常量表达式中使用函数。在编译时对函数进行计算的想法可通过将函数声明为 `constexpr` 来实现。一个 `constexpr` 函数若给定常量表达式作为参数，则函数计算将在编译时完成。例如：

```
constexpr double xscale = 10; // 缩放因子
constexpr double yscale = 0.8;

constexpr Point scale(Point p) { return {xscale*p.x,yscale*p.y}; };
```

假设 `Point` 是一个简单的结构，包含两个成员 `x` 和 `y`，表示一个二维坐标。现在，给 `scale()` 函数一个 `Point` 参数，它返回一个 `Point`，其坐标为输入参数根据 `xscale` 和 `yscale` 缩放后的坐标。例如：

```
void user(Point p1)
{
    Point p2 {10,10};

    Point p3 = scale(p1); // 正确：p3=={100,8}；进行运行时计算
    Point p4 = scale(p2); // p4=={100,8}

    constexpr Point p5 = scale(p1); // 错误：scale(p1) 不是常量表达式
    constexpr Point p6 = scale(p2); // p6=={100,8}

    ...
}
```

一个 `constexpr` 函数和普通函数行为相同，但若在需要一个常量的位置处使用它，则有所不同，此时若传递的参数是常量表达式（如 `p2` 情形），则计算在编译时完成，否则输出错误信息（如 `p1` 情形）。为使该机制可行，要求 `constexpr` 函数必须非常简单，使得编译器（每个符合规范的编译器）能计算它。在 C++11 中，要求 `constexpr` 函数体只能包含一条 `return` 语句（如 `scale()` 所示）；在 C++14 中，还可以写简单的循环语句。一个 `constexpr` 函数不允许有副作用，即它不能改变函数体之外的变量值，当然待赋值或用于初始化的那些变量除外。

以下是违反函数简单性规则的一个函数例子：

```
int glob = 9;

constexpr void bad(int & arg) // 错误：没有返回值
{
    ++arg; // 错误：通过参数修改了某些变量的值
    glob = 7; // 错误：修改了非局部变量
}
```

如果编译器不能确定一个 `constexpr` 函数是否“足够简单”（根据 C++ 标准里的详细规则），该函数会被认为有错误。

8.6 计算顺序

一个程序的计算或称为程序的执行，就是按照语言的规则逐条运行程序中的语句。当这个“执行线程”到达一个变量定义时，变量就会被创建，也就是说编译器会为它分配内存空间，并对其进行初始化。当变量退出其作用域时，将会被销毁，即，原则上它指向的对象会被删除，编译器可把它原来占用的内存作他用。例如：

```

string program_name = "silly";
vector<string> v;                                // v 是全局变量

void f()
{
    string s;                                     // s 是 f 的局部变量
    while (cin>>s && s!="quit") {
        string stripped;                          // stripped 是该循环的局部变量
        string not_letters;
        for (int i=0; i<s.size(); ++i)           // i 的作用域为该语句
            if (isalpha(s[i]))
                stripped += s[i];
            else
                not_letters += s[i];
        v.push_back(stripped);
        // ...
    }
    // ...
}

```

像 `program_name` 和 `v` 这样的全局变量，在 `main()` 的第一条语句执行之前就会被初始化。其生存期直至程序结束，随后会被销毁。它们创建的顺序与定义的顺序相符（`program_name` 先于 `v` 创建），而销毁则按相反的次序（`v` 先于 `program_name` 销毁）。

当有代码调用 `f()` 时，首先会创建 `s`，并将其初始化为空字符串，`s` 的生命期会持续到从 `f()` 返回的时刻。

每次进入 `while` 循环的循环体时，`stripped` 和 `not_letters` 两个变量会被创建。由于 `stripped` 先于 `not_letters` 定义，因此它也先被创建。两个变量的生存期都是到本次循环步的结束为止，在循环条件重新求值之前被销毁，销毁顺序与创建顺序相反（也就是说 `not_letters` 先于 `stripped` 被销毁）。因此，如果我们在遇到字符串“`quit`”之前读入 10 个其他的字符串，`stripped` 和 `not_letters` 将会被创建和销毁 10 次。

每次到达 `for` 循环时，`i` 会被创建。每次退出 `for` 循环时，到达语句 `v.push_back(stripped);` 前，`i` 会被销毁。

请注意，编译器是聪明的家伙，它们获准优化代码，只要得到的结果与我们的目的相符即可。特别是，编译器在分配 / 释放内存方面很聪明，不会不必要地频繁分配 / 释放内存。

8.6.1 表达式计算

表达式中子表达式计算顺序所遵循的规则，是按优化编译器的需求设计的，而不是为了方便程序员。这很不幸，但不管怎样你应该避免复杂的表达式，有一条简单的原则可以帮你远离麻烦：如果你在表达式中改变一个变量的值，不要在同一个表达式中再读或写这个变量。例如：

```

v[i] = ++i;                                // 不要这么做：未定义的计算顺序
v[++i] = i;                                  // 不要这么做：未定义的计算顺序
int x = ++i + ++i;                          // 不要这么做：未定义的计算顺序
cout << ++i << ' ' << i << '\n';      // 不要这么做：未定义的计算顺序
f(++i,++i);                                // 不要这么做：未定义的计算顺序

```

不幸的是，如果你写出这样有问题的代码，并不是所有编译器都能给出警告。这段代码的问题在于，如果你将代码迁移到另外一台计算机，使用另一个编译器，或者改变编译器优

化设置，运行结果不保证一致。不同的编译器确实会对这段代码得到不同结果，所以不要这么编写程序。

特别要注意的是，`=`（赋值符）在表达式中只是又一种运算符而已，并没有特殊的地位，因此不能保证赋值符左边的子表达式在右边的子表达式之前计算。这就是为什么 `v[++i]=i;` 结果是不确定的。

8.6.2 全局初始化

在同一个编译单元中的全局变量（以及名字空间变量，参见 8.7 节）按它们出现的顺序被初始化。例如：

```
// file f1.cpp
int x1 = 1;
int y1 = x1+2;      // y1 变为 3
```

逻辑上这几个变量的初始化在 `main()` 中的代码执行前发生。

 除非是在一些非常特殊的情况下，否则一般来说使用全局变量不是一个好主意。我们已经提到过，程序员没有有效的方法获知程序的哪个部分读或写了一个全局变量（8.4 节）。另一个问题是，在不同编译单元中的全局变量的初始化顺序是不确定的。例如：

```
// file f2.cpp
extern int y1;
int y2 = y1+2;      // y2 为 2 或者 5
```

这段代码存在这样几个问题：使用了全局变量；为全局变量起了很短的名字；对全局变量使用了复杂的初始化。如果文件 `f1.cpp` 中的全局变量先于文件 `f2.cpp` 中的全局变量初始化，那么 `y2` 的初值为 5（这可能是程序员本来所期望的，也是合理的）。但是，如果文件 `f2.cpp` 中的全局变量先于文件 `f1.cpp` 中的全局变量初始化，`y2` 的初值将为 2（因为分配给全局变量的内存空间在变量的复杂初始化前被置为 0）。请避免使用这种代码，并且对复杂的初始化保持足够的警惕，任何不是简单常量表达式的初始化都可以认为是复杂的。

但如果确实需要一个全局变量（或常量），而且需要对它进行复杂的初始化，你又该怎么做呢？一个看起来有道理的例子是，一个用于商务事务的函数库需要一个 `Date` 类型的对象，我们想初始化这个对象：

```
const Date default_date(1970,1,1);      // 默认日期是 1970 年 1 月 1 日
```

我们如何知道 `default_date` 在初始化之前从未被使用过呢？原则上我们不可能知道，因此我们不应该写出这样的代码。一种常用的技术是编写一个函数，返回我们需要的初值。例如：

```
const Date default_date()
{
    return Date(1970,1,1);
}
```

 每当我们调用 `default_date()` 函数，它都会为我们创建一个 `Date` 对象。一般情况下，这种技术已经足够好，但如果需要频繁调用 `default_date()`，而且构造 `Date` 对象代价较高的话，我们更倾向于只构造它一次。可以这样做：

```
const Date& default_date()
{
    static const Date dd(1970,1,1);    // 第一次到达这里时初始化 dd
```

```
    return dd;
}
```

一个静态的局部变量只在函数首次调用时才被初始化（被创建）。注意，这里我们返回一个引用，因此消除了不必要的对象拷贝。特别地，我们返回了一个常量引用，可以防止调用者无意中改变对象的值。本书之前对于如何传递参数的讨论（8.5.6 节），对返回值也是适用的。

8.7 名字空间

在函数中我们用程序块来组织代码（8.4 节）。我们用类来将函数、数据和类型组织到一个类型中（第 9 章）。函数和类都为我们做了如下工作：

- 允许我们定义大量实体，而无须担心它们的名字与程序中其他实体的名字冲突。
- 为我们提供了一个名字，用来访问我们定义的东西。

至此，我们还缺少一种技术，即无须定义一个类型就能将类、函数、数据和类型组织成一个可识别的命名实体。实现这种声明分组功能的 C++ 机制就是名字空间（namespace）。例如，我们希望提供一个包含类 `Color`、`Shape`、`Line`、`Function` 和 `Text` 的绘图库（参见第 18 章）：

```
namespace Graph_lib {
    struct Color {/* ... */};
    struct Shape {/* ... */};
    struct Line : Shape {/* ... */};
    struct Function : Shape {/* ... */};
    struct Text : Shape {/* ... */};
    ...
    int gui_main() {/* ... */}
}
```

很可能其他人也使用了这些名字，但没关系。你可以定义名为 `Text` 的实体，但与我们的 `Text` 没有冲突。`Graph_lib::Text` 是我们定义的类，而你的 `Text` 与之不同。唯一可能有问题的情况就是，你也定义了一个名为 `Graph_lib` 的类或者名字空间，它也包含一个名为 `Text` 的成员。`Graph_lib` 这个名字有点丑，我们选择它的原因是，“漂亮且清晰”的名字 `Graphics` 有很大可能已经被别人用过了。

假设你的 `Text` 是一个文字处理库的一部分。我们用来将绘图功能组织到名字空间 `Graph_lib` 中的思想，也可用来将你的文字处理功能组织到一个叫其他名字（比如 `TextLib`）的名字空间：

```
namespace TextLib {
    class Text {/* ... */};
    class Glyph {/* ... */};
    class Line {/* ... */};
    ...
}
```

如果定义的这两个名字空间都是全局的，我们可能会陷入真正的麻烦之中。假使有人同时使用这两个库，就可能真的遇到名字冲突，如 `Text` 和 `Line`。糟糕的是，如果这两个库都有用户在使用，我们就无法通过修改 `Line`、`Text` 这些名字来避免冲突，否则用户程序也必须修改。为了解决这一问题，我们可以使用名字空间，即我们的 `Text` 用 `Graph_lib::Text` 表示，你的 `Text` 用 `TextLib::Text`。这种由一个名字空间的名字（或一个类名）和一个成员名组合成的名字称为全限定名（fully qualified name）。

8.7.1 using 声明和 using 指令

使用全限定名太繁琐了。例如，C++ 标准库的功能都定义在 std 名字空间中，因此可以按如下方式使用：

```
#include<string>      // 使用 string 库
#include<iostream>     // 使用 iostream 库

int main()
{
    std::string name;
    std::cout << "Please enter your first name\n";
    std::cin >> name;
    std::cout << "Hello, " << name << '\n';
}
```

我们已经见过标准库中的 string 和 cout 无数次了，我们真不希望必须用它们“正确的”全限定名 std::string 和 std::cout 才能访问它们。如果有这么一种方法，能实现“当我说 string，我的意思是 std::string”、“当我说 cout，我的意思是 std::cout”等等，那就好了，就像下面这样：

```
using std::string;      // string 即为 std::string
using std::cout;        // cout 即为 std::cout
// ...
```

这种语法结构称为 using 声明，它与我们常用的人名简称类似：你可以简单地用“Greg”来代表 Greg Hansen，只要屋子里没有其他叫 Greg 的人就没问题。

有时，我们希望可以有一种更强的“简称”用来引用名字空间中的名字：“如果你在本作用域中没有发现某个名字的声明，那么就在 std 中寻找它。”使用 using 指令可以达到这个目的：

```
using namespace std;    // 使 std 中的名字直接可用
```

于是我们得到了下面这种常用的程序风格：

```
#include<string>      // 使用 string 库
#include<iostream>     // 使用 iostream 库
using namespace std;    // 使 std 中的名字直接可用

int main()
{
    string name;
    cout << "Please enter your first name\n";
    cin >> name;
    cout << "Hello, " << name << '\n';
}
```

其中 cin 就表示 std::cin，string 表示 std::string，依此类推。只要你使用 std_lib_facilities.h，你就不需要担心标准库头文件和 std 名字空间了。

 一个一般性的原则是，除非是 std 这种在某个应用领域中大家已经非常熟悉的名字空间，否则最好不要使用 using 指令。过度使用 using 指令带来的问题是，你已经记不清每个名字来自哪里，结果就是你又陷入名字冲突之中。显式使用全限定名或者使用 using 声明就不存在这个问题。因此，将一个 using 指令放在头文件中是一个非常坏的习惯，因为用户就无法避免上述问题。然而，为了简化初学者编写程序，我们确实在 std_lib_facilities.h 中为

std 放置了一条 using 指令，因此我们可以像下面代码这样来写程序：

```
#include "std_lib_facilities.h"

int main()
{
    string name;
    cout << "Please enter your first name\n";
    cin >> name;
    cout << "Hello, " << name << '\n';
}
```

除了 std，我们保证不对任何名字空间使用 using 指令。

简单练习

1. 创建三个文件：my.h、my.cpp 和 use.cpp。头文件 my.h 包含：

```
extern int foo;
void print_foo();
void print(int);
```

源文件 my.cpp 包含 my.h 和 std_lib_facilities.h，定义 print_foo()，此函数用 cout 来打印 foo 的值，my.cpp 中还应定义函数 print(int i)，用 cout 打印 i 的值。

源文件 use.cpp 包含 my.h，定义主函数 main()，主函数中将 foo 的值置为 7，用 print_foo() 打印它，并用 print() 打印整型值 99。注意 use.cpp 并不包含 std_lib_facilities.h，因为它并不直接使用标准库中的功能。

编译并运行这些文件。在 Windows 平台上，你需要将 use.cpp 和 my.cpp 放在同一个项目中，并在 use.cpp 中使用 { char cc; cin>>cc; } 来看到输出结果。提示：你需要包含 iostream 来使用 cin。

2. 编写三个函数 swap_v(int, int)、swap_r(int&, int&) 和 swap_cr(const int&, const int&)。每个函数有如下函数体：

```
{ int temp; temp = a, a=b, b=temp; }
```

其中 a 和 b 是参数名。

尝试用如下代码调用这三个函数：

```
int x = 7;
int y = 9;
swap_?(x,y);           // 将 ? 替换为 v、r 或 cr
swap_?(7,9);
const int cx = 7;
const int cy = 9;
swap_?(cx,cy);
swap_?(7.7,9.9);
double dx = 7.7;
double dy = 9.9;
swap_?(dx,dy);
swap_?(7.7,9.9);
```

哪个函数和调用能编译通过？为什么？对每个编译通过的 swap 调用，在调用过后打印参数的值，检查参数值是否真正被交换了。如果你不理解得到的结果，查阅 8.6 节。

3. 编写一个程序，由一个文件组成，其中包含三个名字空间 X、Y 和 Z，使得如下 main() 函数

数能正确运行：

```
int main()
{
    X::var = 7;
    X::print();           // 打印名字空间 X 中的 var
    using namespace Y;
    var = 9;
    print();              // 打印名字空间 Y 中的 var
    {
        using Z::var;
        using Z::print;
        var = 11;
        print();          // 打印名字空间 Z 中的 var
    }
    print();              // 打印名字空间 Y 中的 var
    X::print();           // 打印名字空间 X 中的 var
}
```

每个名字空间需定义一个名为 var 的变量和一个名为 print() 的函数，该函数用 cout 输出对应的 var 值。

思考题

1. 声明和定义有何区别？
2. 我们如何从语法上区分一个函数声明和一个函数定义？
3. 我们如何从语法上区分一个变量声明和一个变量定义？
4. 对于第 6 章的计算器程序中的函数，为什么不先声明就无法使用？
5. int a; 是一个定义，还是只是一个声明？
6. 为什么说在变量声明时对其初始化是一个好的编程风格？
7. 一个函数声明可以包含哪些内容？
8. 恰当使用缩进有什么好处？
9. 头文件的用处是什么？
10. 什么是声明的作用域？
11. 有几种作用域？请各举一例。
12. 类作用域和局部作用域有何区别？
13. 为什么应该尽量少用全局变量？
14. 传值和传引用有何区别？
15. 传引用和传常量引用有何区别？
16. swap() 是什么？
17. 定义一个函数，它带有一个 vector<double> 类型的传值参数，这样做好吗？
18. 给出一个求值顺序不确定的例子，并说明为什么求值顺序不确定是一个问题。
19. x&&y 和 x||y 分别表示什么？
20. 下面哪些语法结构符合 C++ 标准：函数中的函数，类中的函数，类中的类，函数中的类？
21. 一个活动记录内都包含什么内容？
22. 调用栈是什么？我们为什么需要调用栈？
23. 名字空间的作用是什么？

24. 名字空间和类有何区别？
25. `using` 声明是什么？
26. 为什么应该避免在头文件中使用 `using` 指令？
27. 名字空间 `std` 是什么？

术语

activation record (活动记录)	namespace (名字空间)
argument (参数)	namespace scope (名字空间作用域)
argument passing (参数传递)	nested block (嵌套语句块)
call stack (调用栈)	parameter (参数)
class scope (类作用域)	pass-by-const-reference (传常量引用)
const	pass-by-reference (传引用)
constexpr	pass-by-value (传值)
declaration (声明)	recursion (递归)
definition (定义)	return (return 语句)
extern	return value (返回值)
forward declaration (前置声明)	scope (作用域)
function (函数)	statement scope (语句作用域)
function definition (函数定义)	technicalities (技术细节)
global scope (全局作用域)	undeclared identifier (未定义标识符)
header file (头文件)	using declaration (using 声明)
initializer (初始化程序)	using directive (using 指令)
local scope (局部作用域)	

习题

1. 修改第 7 章的计算器程序，将输入流作为一个显式参数（如 8.5.8 节所示）。并为 `Token_stream` 设计接受 `istream&` 参数的构造函数，这样，当我们解决了如何使用自己的输入流时（比如关联到一个文件），就可以将之用于计算器程序。提示：不要试图拷贝 `istream` 对象。
2. 编写一个函数 `print()`，将一个整型 `vector` 输出到 `cout`。此函数接受两个参数：一个字符串（用于“标记”输出）和一个 `vector`。
3. 创建一个斐波那契数的 `vector`，并用习题 2 中的函数输出这个 `vector`。编写函数 `fibonacci(x,y,v,n)` 来创建 `vector`，其中 `x`、`y` 是 `int` 型，`v` 是 `vector<int>` 类型空 `vector`，`n` 是要放入 `v` 的元素数目，将 `v[0]` 和 `v[1]` 分别设置为 `x` 和 `y`。斐波那契数列是这样一个整型序列，其中每个元素都是前面两个元素之和。例如，以 1 和 2 开始，可以得到斐波那契数列 1, 2, 3, 5, 8, 13, 21, … 你设计的 `fibonacci()` 函数应该以参数 `x` 和 `y` 作为开始，生成如前的斐波那契数列。
4. 计算机中 `int` 型对象能保存的整数的值是有上限的。使用 `fibonacci()` 函数求这个上限的近似值。
5. 编写两个函数，反转一个 `vector<int>` 类型 `vector` 中的元素顺序。例如，将 1、3、5、7、

9 转换为 9、7、5、3、1。第一个反转函数生成一个新 `vector`，其中元素为原 `vector` 的逆序，而原 `vector` 内容不变。另一个反转函数不使用任何其他 `vector`，直接在原 `vector` 中反转元素顺序（提示：`swap`）。

6. 对 `vector<string>` 类型 `vector`，重做习题 5。
7. 读入 5 个名字，存入一个 `vector<string>` 型 `vector name`，然后提示用户输入这些人的年龄，存入一个 `vector<double>` 型 `vector age`。然后打印 5 对 (`name[i], age[i]`)。对名字排序 (`sort(name.begin(), name.end())`)，并输出 (`name[i], age[i]`) 对。此题的难点在于，如何使 `age vector` 中元素的次序与已排序的 `name vector` 中的元素匹配。提示：在排序 `name` 之前，将它复制一份，在排序之后，利用此副本生成顺序正确的 `age` 副本。完成后，重做此题，使之能处理任意数目的姓名和年龄。
8. 编写一个函数，接受两个 `vector<double>` 型参数 `price` 和 `weight`，计算出一个值（“指数”）——所有 `price[i]*weight[i]` 之和。注意，我们必须保证 `weight.size() == price.size()`。
9. 编写一个函数 `maxv()`，返回一个 `vector` 参数中的最大元素。
10. 编写一个函数，接受一个 `vector` 参数，找到最小和最大的元素，并计算均值和中值。不要使用全局变量。或者返回一个 `struct`，包含所有计算结果；或者通过引用参数将所有计算结果返回。这两种返回多个结果的方法中，你更倾向于哪个？为什么？
11. 改进 8.5.2 节中的 `print_until_s()`，并测试它。什么样的测试用例集能更好地测试此程序？请解释原因。然后，编写一个 `print_until_ss()` 函数，它会一直打印，直至第二次看到它的 `quit` 参数。
12. 编写一个函数，接受一个 `vector<string>` 参数，返回一个 `vector<int>`，其每个元素值是对应字符串的长度。此函数还找出最长和最短的字符串，以及字典序第一个和最后一个字符串。为了完成这些工作，你需要用多少个独立的函数？为什么？
13. 我们能声明一个非引用的常量参数吗（如 `void f(const int);`）？这种参数传递方式意味着什么？为什么我们可能需要这种传参方式？为什么我们不应该经常使用这种方式？编写几个小程序来尝试这种传参方式，观察效果。

附言

我们可以将本章（以及下一章）的大部分内容放入附录。然而，你需要了解本章介绍的大部分 C++ 功能，以便后续内容的学习。用这些 C++ 功能可以很快地解决你遇到的很多问题，而这些问题是你承担的一些简单程序设计项目中所必须解决的。因此，为了节约时间，减少混淆，本章系统地介绍了这些内容，而不是让读者去“随机”地访问手册和附录。

类相关的技术细节

记住，做事情要花费时间。

——Piet Hein

在本章中，我们继续关注主要的程序设计工具——C++ 语言。本章主要介绍与用户自定义类型相关的语言技术细节，即类和枚举相关的内容。这些语言特性，大部分是以逐步改进一个 Date 类型的方式来介绍。采用这种方式，我们还可以顺便介绍一些有用的类设计技术。

9.1 用户自定义类型

C++ 语言提供了一些内置类型，如 `char`、`int` 和 `double`（附录 A.8）。对于一个类型，如果编译器无须借助程序员在源码中提供的任何声明，就知道如何表示这种类型的对象以及可以对它进行什么样的运算（例如 `+` 和 `*`），那么我们就称这种类型是内置的（built-in）。

非内置的类型称为用户自定义类型（User-Defined Type，UDT）。用户自定义类型包括每个 ISO 标准 C++ 实现都提供给程序员的标准库类型，如 `string`、`vector` 和 `ostream`（参见第 10 章），也可以是我们为自己创建的类型，如 `Token` 和 `Token_stream`（参见 6.5 节和 6.6 节）。一旦掌握了足够多的必要的语言功能，我们就可以创建图形类型，如 `Shape`、`Line` 和 `Text`（参见第 18 章）。标准库类型很大程度上可以和内置类型一样看作语言的一部分，但我们还是将它们看作用户自定义类型，因为它们和我们自己创建的类型使用了同样的语言功能和技术；标准库的开发者并没有什么特权或者语言工具是你我所不具备的。与内置类型相似，大多数用户自定义类型支持一些运算。例如，`vector` 有 `口` 和 `size()` 运算（参见 4.6.1 节和附录 C.4.8），`ostream` 有 `<<`，`Token_stream` 有 `get()`（参见 6.8 节），`Shape` 有 `add(Point)` 和 `set_color()`（参见 19.2 节）。

为什么要创建自定义类型呢？原因在于编译器不知道我们在程序中使用的所有类型。它也不可能知道，因为有用的类型实在太多了——没有语言设计者或者编译器开发者能预知所有类型。我们每天都创建新的类型。为什么？类型又有什么用？通过类型，我们可以用代码直接、有效地表达我们的思想。当我们编写代码时，理想情况就是能在代码中直接表达思想，这样我们自己、同事以及编译器就能理解代码的含义。当我们想进行算术运算时，`int` 类型会起到很大作用；当我们想处理文本时，`string` 类型会带来很大帮助；当我们想处理计算器的输入时，`Token` 和 `Token_stream` 会起到很大作用。这些类型带来的帮助体现在两个方面：

- 表示（representation）：一个类型“知道”如何表示对象中的数据。
- 运算（operation）：一个类型“知道”可以对对象进行什么运算。

很多编程想法都表现为这种模式：“某些东西”由表示它当前值的一些数据（有时也被称为当前状态（state）），及一组可以应用其上的运算组成。考虑一个计算机文件，一个网页，一

个烤面包机，一台 CD 播放机，一个咖啡杯，一台汽车引擎，一部手机，或是一本电话号码簿；每个对象都可以用一些数据描述，并且支持一组固定的、数量或多或少的标准操作。在每个例子中，操作的结果依赖于对象的数据——“当前状态”。

因此，我们希望在代码中将这样一个“想法”或“概念”表示为一个数据结构加上一组函数。问题是：“如何准确表示？”本章介绍一些在 C++ 中表述概念的一些基本方法，以及涉及的一些语言的技术细节，

 C++ 提供了两类用户自定义数据类型：类和枚举。类是到目前为止最常用的，也是最重要的概念描述机制，因此我们首先把注意力放在类上。类能够在程序中直接地表达概念。一个类（class）是一个（用户自定义）类型，它指出这种类型的对象如何表示，如何创建，如何使用，以及如何销毁（参见 12.5 节）。如果你将某些东西作为一个单独的实体来考虑，那么你可能就应该在程序中定义一个类来表示“这个东西”。这方面的例子有向量、矩阵、输入流、FFT（快速傅里叶变换）、阀门控制器、机械臂、设备驱动、屏幕上的图片、对话框、图形、窗口、温度计读数以及时钟等等。

在 C++（以及大多数现代程序设计语言）中，类是构造大型程序的关键的基本组成部分，对小程序也同样非常有用，这一点我们已经在计算器程序中看到了（第 6 章和第 7 章）。

9.2 类和成员

 一个类就是一个用户自定义类型，由一些内置类型、其他用户自定义类型以及一些函数组成。这些用来定义类的组成部分称为成员（member）。一个类可以有零个或多个成员，例如：

```
class X {
public:
    int m;                                // 数据成员
    int mf(int v) { int old = m; m=v; return old; } // 函数成员
};
```

成员可以有多种类别，大多数要么是数据成员（定义了类对象的表示方法），要么是函数成员（提供类对象之上的运算）。类成员的访问使用符号 *object.member*。例如：

```
X var;          // var 是类型为 X 的变量
var.m = 7;      // 对 var 的数据成员 m 赋值
int x = var.mf(9); // 调用 var 的成员函数 mf()
```

你可以把 *var.m* 读作“*var* 的 *m*”，大多数人念作“*var* 点 *m*”或者“*var* 的 *m*”。一个成员的类型决定了我们可以对它进行什么运算。例如，我们可以读 / 写一个 *int* 成员，可以调用一个函数成员，等等。

一个成员函数，例如类型 *X* 的 *mf()*，不需要使用 *var.m* 这个记号。它可以直接使用成员名字（这里是 *m*）。在成员函数里，一个成员名字表示调用该成员函数的对象中对应的成员。也就是说，函数调用 *var.mf(9)* 中，*mf()* 定义里的 *m* 表示 *var.m*。

9.3 接口和实现

 我们通常把类看作接口加上实现。接口是类声明的一部分，用户可以直接访问它。实现是类声明的另一部分，用户只能通过接口间接访问它。公有的接口用标号 **public:** 标识，实现用标号 **private:** 标识。你可以将一个类声明理解为如下形式：

```

class X { // 类的名字为 X
public:
    // 公有成员:
    // - 用户接口 (可被所有人访问)
    // 函数
    // 类型
    // 数据 (通常最好为 private)
private:
    // 私有成员:
    // - 实现细节 (仅能被该类的成员访问)
    // 函数
    // 类型
    // 数据
};

}

```

类成员默认是私有的，也就是说，如下代码：

```

class X {
    int mf(int);
    // ...
};

}

```

等价于

```

class X {
private:
    int mf(int);
    // ...
};

}

```

因此，下面代码是错误的。

```

X x; // 类型 X 的变量 x
int y = x.mf(); // 错误: mf 是私有的 (不能直接访问)

```

用户不能直接访问一个私有成员，应通过一个公有函数来访问，例如：

```

class X {
    int m;
    int mf(int);
public:
    int f(int i) { m=i; return mf(i); }
};

X x;
int y = x.f(2);

```

我们用私有和公有之间的差别来描述接口（类的用户视图）和实现细节（类的实现者视图）之间的重要区别。下面会逐步给出解释和大量实例，在此我们仅仅指出：如果类只包含数据的话，接口和实现间的区别没有什么意义。C++ 提供了一种很有用的简化功能，可用来描述没有私有实现细节的类。这种语法功能就是结构（struct），一个结构就是一个成员默认为公有属性的类：

```

struct X {
    int m;
    // ...
};

}

```

即为

```
class X {
public:
    int m;
    // ...
};
```

结构主要用于成员可以取任意值的数据结构，也就是说，我们不能定义任何有意义的不变式（参见 9.4.3 节）。

9.4 演化一个类

下面展示如何将一个简单数据结构逐步演化为一个具有私有实现细节和支持函数的类，并解释为什么这么做，我们通过这些内容来说明支持类的语言功能和使用类的基本技术。我们对这些内容的介绍借助于一个非常普通的问题——如何在程序中表示日期（如 1954 年 8 月 14 日）。很显然，很多程序都需要日期，如商业事务程序、天气数据程序、日程表程序、工作记录程序、库存管理程序等等。唯一的问题是，我们如何才能表示它。

9.4.1 结构和函数

如何才能表示一个日期呢？当我们提出这个问题时，很多人会回答：“年、月、日，这样表示如何？”这不是唯一的答案，也不总是最好的答案，但目前对我们来说够用了，这也是我们将要采用的做法。第一个方案是一个简单的结构：

```
// 简单日期结构 Date (太简单? )
struct Date {
    int y;    // 年
    int m;    // 月
    int d;    // 日
};

Date today; // 一个 Date 变量 (命名对象)
```

一个 Date 对象，比如 today，由三个整型简单构成：

Date:	
y:	2005
m:	12
d:	24

这个 Date 结构不存在任何关联的隐藏数据结构，也不能“变出戏法”——而且本章中 Date 的任何一个版本也都是这样。

现在已经有了表示日期的 Date，我们可以对它进行什么操作呢？实际上我们能做任何操作，因为我们可以访问 today（以及任何其他 Date 对象）的成员，并按自己的意愿读写它们。困难在于事情不是真的那么方便，我们想对一个 Date 对象做任何事的话，都必须通过读写其成员的方式来进行，例如：

```
// 设置 today 为 2005 年 12 月 24 日
today.y = 2005;
today.m = 24;
today.d = 12;
```

这样编写程序冗长乏味，而且容易出错。你能看出上面代码中的错误吗？实际上，任何

冗长乏味的东西都容易出错！例如，下面代码有任何意义吗？

```
Date x;  
x.y = -3;  
x.m = 13;  
x.d = 32;
```

很大可能是没有意义的，而且没有人会这么写程序。再考虑下面程序：

```
Date y;  
y.y = 2000;  
y.m = 2;  
y.d = 29;
```

看起来比上一段程序有意义得多，但 2000 年是闰年吗？你确定吗？

较好的方法是设计一些辅助函数，来为我们完成一些最常见的操作。采用这种方法，我们不必一再重复相同的代码，也不必一再犯同样的错误以及查找、修正这些错误。几乎对于每个类型，初始化和赋值都属于最常用的操作。对 Date 来说，增加日期的值是另一个常用操作，于是我们可以编写如下辅助函数：

```
// 辅助函数：  
  
void init_day(Date& dd, int y, int m, int d)  
{  
    // 检查 (y,m,d) 是一个合法的日期  
    // 如果是的话，用其来初始化 dd  
}  
void add_day(Date& dd, int n)  
{  
    // 为日期 dd 加 n 天  
}
```

现在我们可以试着使用 Date 了：

```
void f()  
{  
    Date today;  
    init_day(today, 12, 24, 2005); // 错误！(12 年没有 2005 日)  
    add_day(today, 1);  
}
```

首先，我们注意到这些操作（这里实现为辅助函数）是很有用的。如果我们没有一劳永逸地编写一个日期检查程序的话，检查日期将会是非常困难和乏味的，我们有时可能会忘记写检查代码，从而得到充斥错误的程序。每当定义一个类型，我们都会需要一些针对该类型对象的操作。而到底需要多少个操作，需要什么类型的操作，不同的类型各不相同。我们如何提供这些操作（实现为函数、成员函数或运算符），也是不同的。但只要是决定定义一个类型，我们都要问一下自己，“我们想要为这个类型设计什么样的操作？”

9.4.2 成员函数和构造函数

我们为 Date 设计了一个初始化函数，它提供了重要的日期合法性检查功能。然而，如果我们使用不当的话，日期检查功能将毫无用处。例如，假定我们已经为 Date 定义了输出运算符 << (9.8 节)：

```

void f()
{
    Date today;
    // ...
    cout << today << '\n';           // 使用 today
    // ...
    init_day(today, 2008, 3, 30);
    // ...
    Date tomorrow;
    tomorrow.y = today.y;
    tomorrow.m = today.m;
    tomorrow.d = today.d+1;          // 为 today 加 1 天
    cout << tomorrow << '\n';       // 使用 tomorrow
}

```

这段代码中，我们定义 `today` 后，“忘记了”立即对它进行初始化，而“某人”在我们及时调用 `init_day()` 之前就使用了它。而且“某人”认为调用 `add_day()` 是浪费时间，或许他根本没听说过这个函数，因此他亲手构造了 `tomorrow` 而不是调用 `add_day()`。由于这些情况碰巧发生，这个程序变成了一段问题代码——而且问题非常严重。有时，而且可能是大多数时候，它工作正常，但一些小的改变就可能导致严重的错误。例如，一个未初始化的 `Date` 会产生垃圾输出，而简单地为成员变量 `d` 加 1 来推移日期会成为定时炸弹：当 `today` 表示月底那天时，加 1 操作会导致一个非法的日期。这段“问题严重的代码”最大的问题是，它看起来似乎没什么问题。

上述思考促使我们寻找更好的操作实现方式，我们需要不会被程序员忘记的初始化函数，以及被忽视的可能性很低的操作。实现这些目标的基本技术就是成员函数（member function），即将函数声明于类体内，作为类的成员。例如：

```

// 简单 Date 结构
// 通过构造函数确保初始化
// 提供一些使用便利
struct Date {
    int y, m, d;                  // 年、月、日
    Date(int y, int m, int d);    // 检查日期合法性并进行初始化
    void add_day(int n);         // 为日期增加 n 天
};

```

与类同名的成员函数是特殊的成员函数，称为构造函数（constructor），专门用于类对象的初始化（“构造”）。如果一个类具有需要参数的构造函数，而程序员忘记利用它初始化类对象的话，编译器会捕获这个错误。C++ 提供了一种专用且方便的语法来进行这种初始化：

```

Date my_birthday;                // 错误：my_birthday 未初始化
Date today{12, 24, 2007};         // 哦！运行时错误
Date last{2000, 12, 31};          // 正确（口语风格）
Date next = {2014, 2, 14};        // 也正确（稍显啰嗦）
Date christmas = Date{1976, 12, 24}; // 也正确（冗长风格）

```

试图声明 `my_birthday` 的语句是错误的，因为我们没有指定所需的初值。试图声明 `today` 的语句会编译通过，但构造函数中的检查代码会在运行时捕获非法的日期（12 年 24 月 2007 日，不存在这样的日期）。

定义 `last` 的语句提供了初值——`Date` 的构造函数所需的参数，位置是在紧跟变量名的 {} 列表中。对于一个具有带参数构造函数的类，这是最常见的类变量初始化方式。我们也可以用一种更为啰嗦的方式：显式地创建一个对象（在此处是 `Date{1976, 12, 24}`），然后通过“=”

赋值语法用此初值对变量进行初始化。除非你真的喜欢打字，否则你很快就会厌烦这种方式。

现在，我们可以试着使用新定义的这些变量：

```
last.add_day(1);  
add_day(2);           // 错误：哪个日期？
```

注意，成员函数 `add_day` 必须对特定的 `Date` 对象调用，语法是使用成员访问符号 “.”。我们会在 9.4.4 节介绍如何定义一个成员函数。

在 C++98 中，人们使用圆括号 () 来标记初始化列表，你会看到很多类似下面的代码：

```
Date last(2000,12,31);      // 正确（旧的口语风格）
```

我们倾向于使用 {} 来标记初始化列表，因为它很清晰地表明初始化（构造）过程何时完成，而且这种方式用途更广。比如，还可以用于内置类型的初始化：

```
int x {7};                  // 正确（新的初始化列表风格）
```

或者，可在 {} 列表前加上 “=”：

```
Date next = {2014,2,14};    // 也正确（稍显啰嗦）
```

有些人觉得这种旧方式和新方式的组合可读性更好。

9.4.3 保持细节私有性

现在，我们还有一个问题没有解决：如果有人忘了使用成员函数 `add_day()` 怎么办？如果有人决定直接修改月份怎么办？毕竟，我们“忘了”提供这些功能：

```
Date birthday {1960,12,31};  // 1960 年 12 月 31 日  
++birthday.d;                // 哦！非法日期  
                                // (birthday.d==32 是非法日期)  
  
Date today {1970,2,3};       // 哦！非法日期  
today.m = 14;                 // (today.m==14 是非法日期)
```

只要我们还是将 `Date` 的描述暴露给所有人，那么就会有人（无意地或有意地）把事情搞乱——也就是制造出非法的日期值。例如上面的代码就创建了日历上不存在的日期。这样的非法对象会成为定时炸弹：有人无意间使用非法值只是时间问题，他会得到一个运行时错误，而通常情况会更糟，程序会产生错误的结果。

上述担忧使我们得到如下结论：`Date` 的描述对用户来说应该是不可访问的，除非是通过类中提供的公有成员函数来访问。下面是按这种思想改进后的第一个版本：

```
// 简单 Date 类（控制访问）  
class Date {  
    int y, m, d;          // 年、月、日  
public:  
    Date(int y, int m, int d); // 检查日期合法性并初始化  
    void add_day(int n);    // 为日期增加 n 天  
    int month() { return m; }  
    int day() { return d; }  
    int year() { return y; }  
};
```

使用新版 `Date` 的示例如下：

```

Date birthday {1970, 12, 30};           // 正确
birthday.m = 14;                      // 错误: Date::m 是私有的
cout << birthday.month() << '\n';    // 我们提供了读取 m 的方式

```

 “合法日期”的概念是合法值思想的一个重要特例。我们在设计类型时，设法保证合法值。即，我们隐藏类描述，提供一个创建合法对象的构造函数，所有成员函数的设计也遵循接受合法值、生成合法值的原则。对象的值通常称为状态（state），因此，合法值的思想通常被称为对象的合法状态（valid state）。

我们可以不在每次使用对象时都进行合法性检查，代之以期望没有人到处散布非法值。经验表明，“期望”可以导致“很漂亮的”程序。但是，“很漂亮的”程序偶尔会产生错误的结果或者崩溃，因此编写这样的程序是不能赢得朋友和专业声望的。我们宁愿编写那种可被表明正确性的代码。

 判定合法值的规则称为不变式（invariant）。**Date** 的不变式——“一个 **Date** 对象必须表示过去、现在或将来的某一天”是一个少见的例子，它很难准确表述：我们需要考虑闰年、格里高利历、时区等。但是，如果只是用于特定实际应用的话，我们还是可以给出 **Date** 的不变式的。例如，如果我们分析互联网日志的话，就无须为格里高利历、儒略历或是马雅历而困扰。如果不能想出一个完美的不变式，那我们可能就要处理普通数据。如果是这样，我们可以使用 **struct**。

9.4.4 定义成员函数

到目前为止，我们已经以接口设计者和用户的角度考察了 **Date**，但我们迟早要实现这些成员函数。第一步，我们先给出 **Date** 类声明的一个重新组织过的子集，它将公有接口放在最前面，这也是常用的风格：

```

// 简单 Date 类（有些人习惯将实现细节放在最后）
class Date {
public:
    Date(int y, int m, int d); // 构造函数：检查日期合法性并初始化
    void add_day(int n);     // 为日期增加 n 天
    int month();              // ...
private:
    int y, m, d;             // 年、月、日
};

```

人们把公有接口放在类的开始，是因为接口是大多数人最感兴趣的。理论上，用户无须了解类的实现细节，只需知道接口即可。实际上，我们通常会有好奇心，会快速浏览一下类的实现，看看它是否合理，我们是否能从中学到一些技术。但是，除非是实现者，否则我们会倾向于在公有接口上花更多的时间。编译器并不关心类函数和数据成员的顺序，你愿意以什么样的顺序来声明它们，编译器都能接受。

当我们在类外定义一个成员时，需要指明它是哪个类的成员，这可通过 *class_name::member_name* 方式来实现：

```

Date::Date(int yy, int mm, int dd) // 构造函数
    :y{yy}, m{mm}, d{dd}           // 注意：成员初始化
{
}

void Date::add_day(int n)

```

```

{
    //...
}
int month()           // 噢！忘记了 Date::
{
    return m;          // 不是成员函数，不能访问 m
}

```

符号 :y{yy}, m{mm}, d{dd} 就是类成员初始化的语法，称为初始化列表。当然也可以这样写：

```

Date::Date(int yy, int mm, int dd) // 构造函数
{
    y = yy;
    m = mm;
    d = dd;
}

```

但后一种写法，原则上讲，是先用默认值对成员进行了初始化，然后又对它们进行了赋值。而且这种写法的一个潜在问题是，我们有可能无意地在成员初始化之前使用它们。:y{yy}, m{mm}, d{dd} 这种方式更直接地表达了我们的意图。两种写法之间的区别与下面两段代码之间的区别是一样的：

```

int x;      // 首先定义变量 x
//...
x = 2;      // 后来再给 x 赋值

```

和

```
int x {2}; // 定义并初始化为 2
```

我们也可以直接在类定义中定义成员函数：

```

// 简单 Date 类（有些人习惯将实现细节放在最后）
class Date {
public:
    Date(int yy, int mm, int dd)
        :y{yy}, m{mm}, d{dd}
    {
        //...
    }

    void add_day(int n)
    {
        //...
    }

    int month() { return m; }

    //...
private:
    int y, m, d; // 年、月、日
};

```

我们需要注意的第一点是，这样会使类声明变得大而“凌乱”。例如在此例中，构造函数和 add_day() 的代码会有十几行甚至更长。这使类声明的规模比原来增大几倍，而且使用户难以在实现细节中找到接口。因此，我们不会在类声明中定义大的函数。

但是，看一下 month() 的定义，它比放在类声明之外的版本 Date::month() 要更为直接和简短。对于这种简单的、较小的函数，我们应该考虑直接在类声明中给出其定义。

注意，`month()`可以访问定义在其后（下）的 `m`。实际上，类成员对其他成员的访问并不依赖于成员在类中的声明位置。本书前文介绍的“名字必须在使用之前声明”这一规则，在一个类内的有限作用域中可以放宽。

将成员函数的定义放在类定义内有两个作用：

- ❖ • 函数将成为内联的（`inline`），即编译器为此函数的调用生成代码时，不会生成真正的函数调用，而是将其代码嵌入到调用者的代码中。对于 `month()` 这种所做工作很少，又被频繁调用的函数，这种编译方式会带来很大的性能提升。
- 每当我们对内联函数体做出修改时，所有使用这个类的程序都不得不重新编译。如果函数体位于类声明之外，就不必这样，只在类接口改变时才需要重新编译用户程序。对于大程序来说，函数体改变时无须重新编译程序会是一个巨大的优势。
- 类定义变大了。因此，在成员函数定义中找到成员会变得困难。

显然，我们应遵循如下基本原则：除非你明确需要从小函数的内联中获得性能提升，否则不要将成员函数体放在类声明中。5 行以上代码的函数不会从内联中获益，并且还使得类声明可读性差。对于由超过一、两个表达式组成的函数，本书很少采用内联方式。

9.4.5 引用当前对象

考虑如下使用 `Date` 类的简单代码：

```
class Date {
    // ...
    int month() { return m; }
    // ...
private:
    int y, m, d; // 年、月、日
};

void f(Date d1, Date d2)
{
    cout << d1.month() << ' ' << d2.month() << '\n';
}
```

`Date::month()` 是如何知道第一次被调用时应打印 `d1.m`，第二次被调用时应打印 `d2.m` 呢？再看一下 `Date::month()`，其声明指出它没有任何参数！那么 `Date::month()` 是怎么知道哪个对象在调用它呢？奥妙在这里：每个类成员函数，如 `Date::month()`，都有一个隐式参数，用来识别调用它的对象。因此，在第一次调用中，`m` 会正确地指向 `d1.m`，而在第二次调用中，它指向 `d2.m`。参见 12.10 节，获得更多使用此隐式参数的内容。

9.4.6 报告错误

❖ 当我们发现一个非法日期时，应该做什么呢？检查非法日期的代码应该放在程序中什么位置呢？从 5.6 节我们可以得到第一个问题的答案是“抛出一个异常”，而放置检查代码的位置显然应该是我们最初构造一个 `Date` 对象时。如果没有创建非法的 `Date` 对象，而且成员函数也编写正确，那么我们就永远不会得到具有非法值的 `Date` 对象。因此，我们应该阻止用户创建具有非法状态的 `Date` 对象：

```
// 简单 Date 类（避免非法日期）
class Date {
public:
```

```

class Invalid {};
Date(int y, int m, int d); // 检查日期合法性并初始化
// ...
private:
    int y, m, d; // 年、月、日
    bool is_valid(); // 如果日期合法，则返回 true
};

```

我们将合法性检查代码放到一个独立的函数 `is_valid()` 中，这一方面是因为，从逻辑上讲，合法性检查与初始化就是不同的工作，另一方面是因为，我们可能需要多个构造函数。如你所见，除了私有数据外，我们还可以为类声明私有函数：

```

Date::Date(int yy, int mm, int dd)
    : y(yy), m(mm), d(dd) // 初始化数据成员
{
    if (!is_valid()) throw Invalid(); // 检查合法性
}

bool Date::is_valid() // 如果日期合法，则返回 true
{
    if (m<1 || 12<m) return false;
    // ...
}

```

给出这样的 `Date` 定义后，我们可以写出如下代码：

```

void f(int x, int y)
try {
    Date dxy{2004, x, y};
    cout << dxy << '\n'; // 运算 << 的声明见 9.8 节
    dxy.add_day(2);
}
catch(Date::Invalid) {
    error("invalid date"); // error() 定义见 5.6.3 节
}

```

我们现在知道，`<<` 和 `add_day()` 会获得一个合法的日期作为它们的操作对象。

我们将在 9.7 节完成 `Date` 类的演化，在此之前先介绍几个常用的语句功能，我们需要这些功能来更好地完成 `Date` 类的演化：枚举类型和运算符重载。

9.5 枚举类型

枚举（enumeration，简写为 `enum`）是一种非常简单的用户自定义类型，它指定一个值的集合，这些值用符号常量表示，称为枚举量（enumerator）。下面是一个例子：

```

enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

```

一个枚举定义的“体”就是一个简单的枚举量列表。`enum class` 中的 `class` 表示枚举量在枚举作用域内，也就是说，必须用 `Month::jan` 来表示 `jan`。

你可以为枚举量指定特定的值，就像上面代码为 `jan` 指定值一样。也可以不指定，让编译器选择合适的值。如果让编译器来选择值，它赋予每个枚举量的值为上一个枚举量的值加上 1。因此，上面 `Month` 的定义赋予月份从 1 开始的连续整数。此定义与下面的定义是等价的：

```
enum class Month {
    jan=1, feb=2, mar=3, apr=4, may=5, jun=6,
    jul=7, aug=8, sep=9, oct=10, nov=11, dec=12
};
```

但是，第二种定义一方面冗长乏味，另一方面容易出错。实际上，我们在输入第二个定义时出现了两个错误，经过修改后才得到上面的正确版本。因此，最好还是让编译器来做这种简单的、重复性的“机械性”工作。编译器比我们更擅长这种工作，而且它不会厌烦。

如果我们不初始化第一个枚举量，那么编译器会从 0 开始计数。例如：

```
enum class Day {
    monday, tuesday, wednesday, thursday, friday, saturday, sunday
};
```

这里 `monday==0`，而 `sunday==6`。在实践中，从 0 开始往往是一种好的选择。

我们可以像下面代码那样使用 `Month`：

```
Month m = Month::feb;
Month m2 = feb;           // 错误：feb 不在作用域内
m = 7;                   // 错误：不能对 Month 赋 int 值
int n = m;               // 错误：不能对 int 赋 Month 值
Month mm = Month(7);     // 将整型值转换为 Month (不检查)
```

注意，`Month` 是一个独立的类型，与“构成其基础”的 `int` 型不同。每个 `Month` 值都对应一个相等的整型值，但很多整型值没有相等的 `Month` 值。例如，我们肯定希望下面的初始化失败：

```
Month bad = 9999;         // 错误：不能将 int 转换为 Month
```

⚠ 如果你非要坚持使用 `Month(9999)` 这种语法，那么一切结果由你自己负责！很多情况下，如果程序员明确地坚持做一些可能很蠢的事情，C++ 不会设法阻止，毕竟程序员可能更清楚自己在做什么。值得一提的是，不能使用 `Month{9999}` 记号，因为这里只允许能用于初始化 `Month` 的值这么做，而 `int` 不可以。

遗憾的是，我们不能为枚举类型定义一个构造函数来检查初始值，但编写一个简单的检查函数还是很容易的：

```
Month int_to_month(int x)
{
    if (x<int(Month::jan) || int(Month::dec)<x) error("bad month");
    return Month(x);
}
```

这里记号 `int(Month::jan)` 表示得到 `Month::jan` 对应的 `int` 值。有了这个函数，我们就可以这样使用：

```
void f(int m)
{
    Month mm = int_to_month(m);
    // ...
}
```

我们能用枚举类型做什么呢？原则上，枚举类型可以用于任何需要一组相关的命名整型常量的地方。当我们想表示选项（如上、下；是、否、也许；开、关；北、东北、东、东南、南、西南、西、西北等）或表示不同值（如红、蓝、绿、黄、栗、深红、黑）时，就属于这

种情况。

9.5.1 “平坦” 枚举

使用 `enum class` 定义的枚举也被称为是“作用域枚举”(scoped enumeration)，除此之外，还有一种称为“平坦”(plain) 枚举的定义，和作用域枚举的区别是，平坦枚举将它的枚举量都隐式导出到枚举类型所在的作用域里，并且可以隐式转换到 `int` 型。例如：

```
enum Month {                                // 注意：此处没有“class”
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;                         // 正确：feb 在该作用域里
Month m2 = Month::feb;                  // 同样正确
m = 7;                                 // 错误：不能对 Month 赋 int 值
int n = m;                            // 正确：可以对 int 赋 Month 值
Month mm = Month(7);                   // 转换 int 到 Month (不检查)
```

很明显，平坦枚举没有作用域枚举严格。平坦枚举的枚举量能“污染”枚举类型所在的作用域。有时候这很方便，有时也会导致混乱。例如，当你试图同时使用 `Month` 和 `iostream` 格式化机制(11.2.1节)时，会发现代表12月(December)的 `dec` 和代表十进制(decimal)的 `dec` 有冲突。

类似地，允许枚举值转换为 `int` 值在使用上会很方便(当我们想转换枚举量到 `int` 时无须再显式进行)，但有时也会导致诧异。例如：

```
void my_code(Month m)
{
    If (m==17) do_something();           // 17月吗？
    If (m==monday) do_something_else(); // 将 month 和 Monday 进行比较吗？
}
```

如果 `Month` 是 `enum class`，两个 `if` 条件都不会被成功编译。如果 `monday` 是平坦枚举量(而不是作用域枚举量)，那么 `month` 和 `Monday` 的比较是允许的，但很可能这会导致意外结果。

我们倾向于使用更简单、更安全的作用域枚举类型，少用平坦枚举类型，但是在旧的代码中有很多平坦枚举量，这是因为 `enum class` 是 C++11 中新出现的功能。

9.6 运算符重载

你可以在类或枚举对象上定义几乎所有 C++ 运算符，这通常称为运算符重载(operator overloading)。这种机制用于为用户自定义类型提供习惯的符号表示方法。如下例：

```
enum class Month {
    Jan=1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
};

Month operator++(Month& m)                // 前缀递增运算符
{
    m = (m==Dec) ? Jan : Month(int(m)+1); // “绕回”
    return m;
}
```

其中`?:`是“算术 if”运算符：当 `(m==Dec)` 时 `m` 的值变为 `Jan`，否则 `m` 的值为 `Month(int(m)+1)`。这是对十二月后继月份“绕回”一月这一特性的一种非常简洁的描述方法。现在，我们可以

像如下代码一样使用 Month 类型：

```
Month m = Sep;
++m;           // m 变为 Oct
++m;           // m 变为 Nov
++m;           // m 变为 Dec
++m;           // m 变为 Jan (“绕回”)
```

你可能觉得增加 Month 对象值这样的操作没那么常用，不至于设计一个专门的运算符。可能确实是这样，那么输出运算符又如何呢？如下代码定义了一个输出运算符：

```
vector<string> month_tbl;

ostream& operator<<(ostream& os, Month m)
{
    return os << month_tbl[int(m)];
}
```

这里假定 month_tbl 已经在其他位置进行了初始化，每个数组元素保存了对应月份的恰当的名字，如 month_tbl[int(Month::mar)] 的值为字符串 “ March ” 或其他合适的名字；参见 10.11.3 节。

 你可以为自己的类型重新定义几乎所有的 C++ 运算符，如 +、-、*、/、%、[]、()、^、!、&、<、<=、>、>= 等等。不能定义新的运算符，你可能想在程序中把 ** 或 \$= 作为运算符，但 C++ 不允许你这样做。而且，重载运算符时，运算对象数目也必须与原来一样。例如，你可以定义一元运算符 -，但不能定义一元的 <=（小于等于），你可以定义二元运算符 +，但不能定义二元的 !（非）。原则上，C++ 允许你对自定义类型使用已有的语法，但不允许扩展语法。

一个重载的运算符必须作用于至少一个用户自定义类型的运算对象：

```
int operator+(int,int); // 错误：不能重载内置类型 + 运算符
Vector operator+(const Vector&, const Vector &); // 正确
Vector operator+=(const Vector&, int); // 正确
```

 一般性的原则是：除非你真正确定重载运算符能大大改善代码，否则不要为你的类型定义运算符。而且，重载运算符应该保持其原有意义：+ 就应该是加法，二元运算符 * 就应该表示乘法，[] 表示元素访问，() 表示调用，等等。这只是建议，并不是 C++ 语言规则，但这是一个有益的建议：按习惯使用运算符，例如 + 只用做加法，对我们理解程序会有极大的帮助。毕竟，这种习惯用法源于人们千百年来使用数学符号的经验。相反，含混不清的运算符和不符合常规的使用方式是混乱和错误之源。我们不再过多阐述这一点，相反，在后面的章节中，我们只是在合适的地方使用运算符重载。

注意，不像大家一般猜想的那样，重载最多的运算符不是 +、-、*、/，而是 =、==、!=、<、[]（下标运算符）及 ()（函数调用运算符）。

9.7 类接口

我们已经讨论过，类的公有接口和实现部分应该分离。只要为那些“旧式简单数据”类型保留着 struct 功能，不同意接口和实现分离原则的专业人员就会很少。但是，我们如何来设计一个好的接口呢？好的公有接口和乱糟糟的接口之间有什么区别呢？回答这个问题必须借助实例，但我们仍能列出一些一般原则，它们在 C++ 中都有支持：



- 保持接口的完整性。
- 保持接口的最小化。
- 提供构造函数。
- 支持（或者禁止）拷贝（参见 19.2.4 节）。
- 使用类型来提供完善的参数检查。
- 识别不可修改的成员函数（参见 9.7.4 节）。
- 在析构函数中释放所有资源（参见 12.5 节）。

也请参考 5.5 节（如何检测及报告运行时错误）。

前两条原则可以归结为“保持接口尽可能小，但不要更小了”。我们希望接口尽量小，是因为小的接口易于学习和记忆，而实现者也不会为不必要的和很少使用的功能浪费大量时间。小的接口还意味着有错误发生时，我们只需检查很少的函数来定位错误。平均来看，公有成员函数越多，查找 bug 就越困难——调试带有公有数据的类是非常复杂的，不要让我们陷入其中。当然，前提还是要保证完整性，否则接口就没有用处了。如果一个接口无法完成我们真正需要做的全部工作，我们是不会使用它的。

下面我们讨论其他一些话题，这些话题更为具体，直接对应 C++ 语言功能。

9.7.1 参数类型

当我们在 9.4.3 节中为 Date 定义构造函数时，使用了三个整型作为其参数。这会带来一些问题：

```
Date d1 {4,5,2005}; // 噢！4 年 2005 日  
Date d2 {2005,4,5}; // 4 月 5 日还是 5 月 4 日？
```

第一个问题（非法的日期）比较容易处理，在构造函数中进行检测即可。但是，第二个问题（月和日的混淆），通过用户编写的检测代码是无法查找到的。这个问题是由于人们书写月和日的习惯不同而造成的：例如，4/5 在美国表示 4 月 5 日，但在英国表示 5 月 4 日。我们不能指望不遇到这个问题，必须采取其他手段解决它。一种明显的解决方案是使用 Month 类型：

```
enum class Month {  
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec  
};  
  
// 简单 Date 类（使用 Month 类型）  
class Date {  
public:  
    Date(int y, Month m, int d); // 检查日期合法性并初始化  
    // ...  
private:  
    int y; // 年  
    Month m;  
    int d; // 日  
};
```

如果使用了 Month 类型，当我们颠倒了月和日这两个参数时，编译器就会捕获这个问题。而且，使用一个枚举类型作为月的类型，令我们可以使用符号名来表示月份，这样的代码通常比直接使用数字更易读，因而也更不容易出错：

```

Date dx1 {1998, 4, 3};           // 错误：第 2 个参数不是 Month
Date dx2 {1998, 4, Month::mar};    // 错误：第 2 个参数不是 Month
Date dx2 {4, Month::mar, 1998};    // 哦！运行时错误：该月第 1998 天
Date dx2 {Month::mar, 4, 1998};    // 错误：第 2 个参数不是 Month
Date dx3 {1998, Month::mar, 30};    // 正确

```

这段代码中，编译器帮我们避免了很多“意外”。注意代码中使用了枚举量 `mar` 的限定名 `Month::mar`。我们没有用 `Month.mar`，因为 `Month` 不是一个对象（而是一个类型），而 `mar` 也不是一个数据成员（而是一个枚举量——一个符号常量）。我们在类名、枚举名或名字空间名（参见 8.7 节）后使用 `::`，而在对象名后使用 `.`（点）。

如果可以选择，我们宁可在编译时将错误捕获，而不要留在运行时。我们更希望由编译器找到错误，而不是由我们自己来寻找到底是代码哪个位置发生了问题。而且，如果错误可以在编译时被捕获，我们就不需要编写并执行检查代码。

考虑这么一个问题：我们能捕获颠倒日与年的情况吗？当然是可以的，但是解决方案不像处理日与月的颠倒那么简单、优美。毕竟，像公元 4 年这样的年份也是合法的，我们的方案必须能表示这样的年份。即使将日期限定为近代，需要表示的年份也实在是太多了，无法定义一个枚举来描述所有这些年份。

可能我们能做到的最好程度（在不了解很多 `Date` 用途的情况下）像下面代码这样：

```

class Year {                                // 年份范围是 [min:max]
    static const int min = 1800;
    static const int max = 2200;
public:
    class Invalid {};
    Year(int x) : y{x} { if (x<min || max<=x) throw Invalid{}; }
    int year() { return y; }
private:
    int y;
};

class Date {
public:
    Date(Year y, Month m, int d);          // 检查日期合法性并初始化
    // ...
private:
    Year y;
    Month m;
    int d; // 日
};

```

现在，我们可以在编译时避免年份颠倒的错误了：

```

Date dx1 {Year{1998}, 4, 3};           // 错误：第 2 个参数不是 Month
Date dx2 {Year{1998}, 4, Month::mar};    // 错误：第 2 个参数不是 Month
Date dx2 {4, Month::mar, Year{1998}};    // 错误：第 1 个参数不是 Year
Date dx2 {Month::mar, 4, Year{1998}};    // 错误：第 2 个参数不是 Month
Date dx3 {Year{1998}, Month::mar, 30};    // 正确

```

当然，下面这种怪异的、不太可能出现的错误，编译时仍无法捕获：

```
Date dx2 {Year{4}, Month::mar, 1998};    // 运行时错误：抛出异常 Year::Invalid
```

做这些额外的工作、引入这些新的符号来进行年份的检查，是否值得呢？这自然取决于你用 `Date` 解决什么问题。但在本书中，我们认为没有必要这么做，因此后面不会使用 `Year` 类。

当我们编程时，应该一直问自己：对于给定的应用，什么是足够好的解决方案？我们通

常不会奢侈到，在已经得到一个足够好的方案后，还会“无止境地”去追求最佳方案。继续追寻下去的话，我们甚至可能得到一些非常复杂，但却比最初的简单方案更差的方案。人们常说的“至善者善之敌”（“The best is the enemy of the good”，伏尔泰）就是这个意思。

注意 `min` 和 `max` 定义中对 `static const` 的使用。这就是我们在类中定义整型符号常量的方法。我们使用 `static` 限定一个类成员，可以保证它在整个程序中只有一份拷贝，而不是每个类对象都有一份。在此处，由于初始化值是常量表达式，我们可以用 `constexpr` 来替代 `const`。

9.7.2 拷贝

编程中总是要创建对象的，也就是说，我们总是要考虑初始化和构造函数。构造函数可能是最重要的类成员：为了编写构造函数，我们必须确定初始化一个对象时应该做什么，以及什么样的值是合法值（不变式是什么）。单纯地考虑初始化工作，会帮助你在设计构造函数时避免错误。

下一个经常要考虑的问题是：我们需要拷贝对象吗？如果可以，如何拷贝呢？

对于 `Date` 或 `Month`，答案是：我们显然需要拷贝这两种类型的对象；而这两种类型的对象的拷贝的含义很简单——只要复制所有成员即可。实际上，这正是默认情形。只要你不特别声明，编译器就会正确地做到上述效果。例如，如果你将一个 `Month` 对象作为初始化值和赋值运算的右部，编译器就会完成其所有成员的拷贝：

```
Date holiday {1978, Month::jul, 4};      // 初始化
Date d2 = holiday;
Date d3 = Date{1978, Month::jul, 4};
holiday = Date{1978, Month::dec, 24};    // 赋值
d3 = holiday;
```

这段代码已经完全按我们的期望工作了。用 `Date{1978, Month::dec, 24}` 可以创建一个正确的未命名 `Date` 对象，你可以用它来做一些适当的工作。例如：

```
cout << Date{1978, Month::dec, 24};
```

这里对构造函数的使用很像类作为类型的字面值常量。通常，当我们需要定义一个只使用一次的变量或常量时，这是一种很方便的方法。

如果我们需要拷贝操作的含义与默认情况不同，应该怎么做呢？可以定义自己的拷贝函数（参见 13.3 节），或者将拷贝构造函数和拷贝赋值运算符描述为 `delete`（参见 19.2.4 节）。

9.7.3 默认构造函数

未初始化的变量可能会成为错误之源。为了解决这个问题，我们可以用构造函数来保证类的每个对象都被初始化。例如，我们定义了构造函数 `Date::Date(int, Month, int)` 来保证每个 `Date` 对象都会被正确地初始化。这意味着程序员必须提供三个类型正确的参数。例如：

<code>Date d0;</code>	<code>// 错误：没有初始化值</code>
<code>Date d1 {};</code>	<code>// 错误：空的初始化值</code>
<code>Date d2 {1998};</code>	<code>// 错误：参数太少</code>
<code>Date d3 {1,2,3,4};</code>	<code>// 错误：参数太多</code>
<code>Date d4 {1,"jan",2};</code>	<code>// 错误：参数类型错误</code>
<code>Date d5 {1,Month::jan,2};</code>	<code>// 正确：使用 3 个参数的构造函数</code>
<code>Date d6 {d5};</code>	<code>// 正确：使用拷贝构造函数</code>

注意，虽然我们为 `Date` 定义了一个需要三个参数的构造函数，但是通过赋值运算直接拷贝还是没有问题的。

很多类都能很好地理解默认值，也就是说，它们能解决这个问题：“如果我没有为对象提供一个初始值，那么它应该具有什么值？”例如：

```
string s1; // 默认值：空字符串
vector<string> v1; // 默认值：空 vector，没有元素
```

这些代码就像注释所描述的那样工作，这看起来很合理。之所以 `vector` 和 `string` 类能支持这样的特性，是因为它们都有默认构造函数（default constructor），可以隐含地进行所需的初始化工作。

对于类型 `T`，符号 `T0` 表示默认值，这是通过定义默认构造函数实现的，因此，我们可以写出下面这样的代码：

```
string s1 = string{}; // 默认值：空字符串
vector<string> v1 = vector<string>{}; // 默认值：空 vector，没有元素
```

但是，我们更倾向于采用下面这种等价的，但更“口语化”的语法形式：

```
string s1; // 默认值：空字符串
vector<string> v1; // 默认值：空 vector，没有元素
```

对于内置类型，如 `int` 和 `double` 来说，默认构造函数的结果为 0，即，`int0` 是 0 的一种复杂描述，而 `double0` 是 0.0 的一种啰嗦的说法。

使用默认构造函数不只是形式上的问题，它有着更深层次的重要作用。设想一下，若我们有一个未初始化的 `string` 或 `vector` 对象：

```
string s;
for (int i=0; i<s.size(); ++i) // 噢！循环次数未知
    s[i] = toupper(s[i]); // 噢！读写一个随机的内存位置

vector<string> v;
v.push_back("bad"); // 噢！写入一个随机地址
```

如果 `s` 和 `v` 的值真的是未定义的话，我们就完全不知道它们包含多少个元素或者（采用一些常用的实现技术，参见 12.5 节）无法知道这些元素存放在哪里。其结果就是我们可能会访问随机地址——这会导致最麻烦的一类错误。基本上，没有构造函数的话，我们就无法建立一个不变式——也就不能保证变量中的值是合法的（9.4.3 节）。我们必须坚持对变量进行初始化。我们可以坚持使用初始化代码，像下面代码这样：

```
string s1 = "";
vector<string> v1 {};
```

但是，这种方法并不是非常完美。对于 `string` 类型，“”表示“空字符串”是显然的。对于 `vector` 类型，`{}` 可以很好地表示空 `vector`。然而，对于很多类型来说，找到一个值，能合理地表示默认值，并不容易。因此，更好的方法是定义一个构造函数，不需要程序员显式提供初始化代码，就能创建对象。这种构造函数不接受参数，我们称之为默认构造函数（default constructor）。

例如，对于日期来说，就不存在一个显然的默认值。这就是为什么到目前为止还没有为 `Date` 定义一个默认构造函数的原因，现在我们为它定义一个（只是说明我们可以这样做而已）：

```
class Date {
public:
```

```

//...
Date(); // 默认构造函数
//...
private:
    int y;
    Month m;
    int d;
};

}

```

我们需要挑选一个默认日期，21世纪的第一天可能是个合理的选择：

```

Date::Date()
    :y(2001), m{Month::jan}, d{1}
{
}

```

若不将成员默认值放在构造函数里，我们也可以将它们放在成员本身声明处：

```

class Date {
public:
    //...
    Date(); // 默认构造函数
    Date(year, Month, day);
    Date(int y); // y 年 1 月 1 日
    //...
private:
    int y {2001};
    Month m {Month::jan};
    int d {1};
}

```

这样的话，每个构造函数都可以使用默认值。例如：

```

Date::Date(int y) // y 年 1 月 1 日
    :y(yy)
{
    if (!is_valid()) throw Invalid{}; // 检查合法性
}

```

因为 **Date(int)** 没有显式初始化月 (m) 和日 (d)，所以隐式使用了指定的初始化值 (Month::jan 和 1)。一个类成员在声明时指定的初始化值被称为类内初始化值。

如果不想把默认值写入构造函数的代码中，我们可以使用一个常量（或一个变量）。为了避免使用全局变量带来的初始化等问题，我们使用 8.6.2 节中介绍的技术：

```

const Date& default_date()
{
    static Date dd {2001, Month::jan, 1};
    return dd;
}

```

这里使用了 **static**，这样变量 dd 就不会每次调用 **default_date()** 时都被创建，它只在第一次调用 **default_date()** 时被创建并被初始化。有了 **default_date()** 函数，为 **Date** 创建一个默认构造函数就很简单了：

```

Date::Date()
    :y(default_date().year()),
    m(default_date().month()),
    d(default_date().day())
{
}

```

注意，默认构造函数无须检查对象值，因为 `default_date` 的构造函数已经做过了。有了 `Date` 的默认构造函数，我们就可以定义 `Date` 数组了：

```
vector<Date> birthdays(10); // 10 个元素，均为默认值 Date()
```

如果没有默认构造函数，我们可能不得不显式这样做：

```
vector<Date> birthdays(10, default_date()); // 10 个 Dates 默认值

vector<Date> birthdays2 = { // 10 个 Dates 默认值
    default_date(), default_date(), default_date(), default_date(), default_
    date(),
    default_date(), default_date(), default_date(), default_date(), default_
    date()
};
```

当需要指定 `vector` 的元素个数时，我们使用圆括号（）而不是 {} 初始化列表记号，这可避免 `vector<int>` 情形时的混淆（见 13.2 节）。

9.7.4 const 成员函数

对于有些变量，我们希望它们可以被改变——这也是为什么我们称之为“变量”的原因。但对于另外一些变量，我们则不希望改变它们。即，我们想用“变量”表示的实际上是不变量。我们通常称它们为常量（constant，或者简写为 `const`）。考虑下面代码：

```
void some_function(Date& d, const Date& start_of_term)
{
    int a = d.day(); // 可以
    int b = start_of_term.day(); // 应该可以（为什么？）
    d.add_day(3); // 可以
    start_of_term.add_day(3); // 错误
}
```

在这里，我们希望 `d` 是可变的，`start_of_term` 是不可变的，而 `some_function()` 将不被允许对 `start_of_term` 进行更改。编译器是如何知道这些的呢？这是因为我们将 `start_of_term` 定义为 `const`，从而使编译器获得了上述信息。好了，我们达到了预期的目的，但是，为什么用 `day()` 来读取 `start_of_term` 的成员 `day` 是被允许的呢？实际上，根据前面给出的 `Date` 的定义，`start_of_term.day()` 是错误的，因为编译器不知道 `day()` 是否修改了对象的日期。我们没有给出过这方面的任何信息，因此编译器应该假定 `day()` 有可能改变日期，并报告一个错误。

 我们可以将类操作划分为两类——可更改和不可更改，这样就可以解决这个问题了。这个语言特性对于我们深入理解类是非常重要的，而且它也具有很重要的实践意义：不修改对象的操作可以在常量对象上调用。如下例：

```
class Date {
public:
    ...
    int day() const; // 常量成员：不改变对象
    Month month() const; // 常量成员：不改变对象
    int year() const; // 常量成员：不改变对象

    void add_day(int n); // 非常量成员：可以改变对象
    void add_month(int n); // 非常量成员：可以改变对象
    void add_year(int n); // 非常量成员：可以改变对象

private:
    int y; // 年
```

```

Month m;
int d; // 日 // 错误: cd 是常量
};

Date d {2000, Month::jan, 20};
const Date cd {2001, Month::feb, 21};

cout << d.day() << " - " << cd.day() << '\n'; // 正确
d.add_day(1); // 正确
cd.add_day(1); // 错误: cd 是常量

```

在一个成员函数声明中，我们将 `const` 放置参数列表右边，就表示这个成员函数可以在一个常量对象上调用。一旦将一个成员函数声明为 `const`，编译器会帮助我们保证这个成员函数不会修改对象。例如：

```

int Date::day() const
{
    ++d; // 错误: 试图从常量成员函数中改变对象
    return d;
}

```

当然，通常我们不会故意这么做。但我们可能会无意中这么做，特别是当代码非常复杂时，而编译器可以保证避免这样的问题。

9.7.5 类成员和“辅助函数”

当我们试图最小化类接口时（在保证完整性的前提下），不得不忽略大量有用的操作。如果一个函数可以简单、优美、高效地实现为一个独立函数时（即实现为非成员函数），就应该将它的实现放在类外。采用这种方式，函数中的错误就不会直接破坏类对象中的数据。不访问类表示是很重要的，因为常用的 debug 技术是“首先排查惯犯”，即，当类出现问题时，我们首先检查直接访问类表示的函数：几乎可以肯定这是这类函数导致的错误。如果这类函数只有十几个而不是 50 个的话，我们当然会很高兴。

`Date` 类有 50 个成员函数！你一定认为我们是在开玩笑。但我们没有：几年前我调查了一些商用的 `Date` 库，发现这些库中充斥着像 `next_Sunday()`、`next_workday()` 这样的函数。对于一个设计目标更倾向于方便用户使用，而不是易于理解、实现和维护的类来说，有 50 个成员函数并不过分。

另一点值得注意的是，如果类表示改变，只有直接访问类表示的函数才需要重写。这是保持接口最小化的另一个重要原因。在 `Date` 例子中，我们可能会觉得用一个整数表示自 1900 年 1 月 1 日至今的天数，比用（年，月，日）的形式来表示好得多。如果做出这样的改变，只有成员函数需要进行修改。

下面是一些辅助函数（helper function）的例子：

```

Date next_Sunday(const Date& d)
{
    // 使用 d.day()、d.month()、d.year 访问 d
    // 创建新的 Date 并返回
}
Date next_weekday(const Date& d) /* ... */

bool leapyear(int y) /* ... */

bool operator==(const Date& a, const Date& b)

```

```

{
    return a.year()==b.year()
        && a.month()==b.month()
        && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}

```

 辅助函数还被称为便利函数、帮助函数等。这类函数和其他非成员函数在逻辑上是有区别的——辅助函数是一种设计思想，而不是一种编程概念。辅助函数通常接受一个类对象作为其参数，它就是为这个类做辅助工作。当然也有例外，参考 leapyear()。我们通常用名字空间来区分一组辅助函数，参见 8.7 节：

```

namespace Chrono {
    enum class Month /* ... */;
    class Date /* ... */;
    bool is_date(int y, Month m, int d); // 当日期合法时返回 true
    Date next_Sunday(const Date& d) /* ... */;
    Date next_weekday(const Date& d) /* ... */;
    bool leapyear(int y) /* ... */; // 见习题 10
    bool operator==(const Date& a, const Date& b) /* ... */
    //...
}

```

请注意 == 和 != 函数，它们是典型的辅助函数。对于很多类来说，== 和 != 具有明显的意义，但它们又不是对所有类都有意义，因此编译器无法像处理拷贝构造函数和赋值运算符那样为你定义默认的 == 和 != 函数。

还请注意我们引入了一个辅助函数 is_date()。这个函数代替了 Date::is_valid()，因为检查一个日期是否合法很大程度上与 Date 的表示是无关的。例如，我们无须知道 Date 对象是如何表示的，就可判断“2008 年 1 月 30 日”是合法的，“2008 年 2 月 30 日”是不合法的。还有一些日期相关的问题可能依赖于表示方法（例如，我们可以表示“1066 年 1 月 30 日”吗？），但这可以由 Date 的构造函数来处理（如果需要的话）。

9.8 Date 类

现在，让我们将这一章介绍的内容组合在一起，看看能设计出什么样的 Date 类来。下面代码中若函数体只是一个“...”的注释，说明具体实现很复杂（请不要现在就尝试实现它们）。首先，我们将声明放在头文件 Chrono.h 中：

```

// 文件 Chrono.h

namespace Chrono {

enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

class Date {
public:
    class Invalid {}; // 作为异常抛出
}

```

```

Date(int y, Month m, int d); // 检查日期合法性并初始化
Date(); // 默认构造函数
// 默认拷贝操作是可用的

// 不改变对象的操作：
int day() const { return d; }
Month month() const { return m; }
int year() const { return y; }

// 改变对象的操作：
void add_day(int n);
void add_month(int n);
void add_year(int n);

private:
    int y;
    Month m;
    int d;
};

bool is_date(int y, Month m, int d); // 当日期合法时返回 true
bool leapyear(int y); // 当 y 是闰年时返回 true

bool operator==(const Date& a, const Date& b);
bool operator!=(const Date& a, const Date& b);

ostream& operator<<(ostream& os, const Date& d);
istream& operator>>(istream& is, Date& dd);

}
```

// Chrono

成员函数的定义在 Chrono.cpp 中：

```

// Chrono.cpp
#include "Chrono.h"

namespace Chrono {
// 成员函数定义：

Date::Date(int yy, Month mm, int dd)
    : y{yy}, m{mm}, d{dd}
{
    if (!is_date(yy, mm, dd)) throw Invalid();
}

const Date& default_date()
{
    static Date dd {2001, Month::jan, 1}; // 21 世纪的开始
    return dd;
}

Date::Date()
    : y{default_date().year()},
    m{default_date().month()},
    d{default_date().day()}
{
}

void Date::add_day(int n)

```

```

{
    // ...
}

void Date::add_month(int n)
{
    // ...
}

void Date::add_year(int n)
{
    if (m==feb && d==29 && !leapyear(y+n)) {      // 小心闰年!
        m = mar;                                // 用 3 月 1 日代替 2 月 29 日
        d = 1;
    }
    y+=n;
}
// 辅助函数:

bool is_date(int y, Month m, int d)
{
    // 假设 y 是合法的

    if (d<=0) return false;           // d 必须为正
    if (m<Month::jan || Month::dec<m) return false;

    int days_in_month = 31;          // 大多数月份都有 31 天

    switch (m) {
        case Month::feb:           // 2 月天数可变
            days_in_month = (leapyear(y))?29:28;
            break;
        case Month::apr: case Month::jun: case Month::sep: case Month::nov:
            days_in_month = 30;      // 其余的月份都是 30 天
            break;
    }

    if (days_in_month<d) return false;

    return true;
}

bool leapyear(int y)
{
    // 见习题 10
}

bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
        && a.month()==b.month()
        && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b)
{
    return !(a==b);
}

ostream& operator<<(ostream& os, const Date& d)
{
}

```

```

return os << '(' << d.year()
    << ',' << d.month()
    << ',' << d.day() << ')';
}

istream& operator>>(istream& is, Date& dd)
{
    int y, m, d;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;
    if (ch1!= '(' || ch2!= ',' || ch3!= ',' || ch4!= ')') { // 噢！格式错误
        is.clear(ios_base::failbit); // 设置 failbit
    }
    return is;
}

dd = Date(y, Month(m), d); // 更新 dd

return is;
}

enum class Day {
    sunday, monday, tuesday, wednesday, thursday, friday, saturday
};

Day day_of_week(const Date& d)
{
    // ...
}
Date next_Sunday(const Date& d)
{
    // ...
}

Date next_weekday(const Date& d)
{
    // ...
}
} // Chrono

```

函数 `>>` 和 `<<` 的实现将在 10.8 节和 10.9 节中详细解释。

简单练习

本练习的目的就是使一系列 `Date` 版本能正常工作。对每个版本，请定义一个名为 `today` 的 `Date` 对象，初值为 1978 年 6 月 25 日。然后，定义一个名为 `tomorrow` 的 `Date` 对象，通过拷贝 `today` 对其赋值，随后使用 `add_day()` 将其向后推移一天。最后，使用 9.8 节中定义的 `<<` 输出 `today` 和 `tomorrow`。

合法日期的检查可以简单实现。可以先忽略闰年的情形，但应保证不接受 [1, 12] 范围之外的月份和 [1, 31] 范围之外的日期。对每个版本至少用一个非法日期来测试（如 2004 年 13 月 -5 日）。

1. 9.4.1 节中的版本。
2. 9.4.2 节中的版本。

3. 9.4.3 节中的版本。
4. 9.7.1 节中的版本。
5. 9.7.4 节中的版本。

思考题

1. 本章中所描述的类的两个组成部分是什么？
2. 一个类中，接口和实现的区别是什么？
3. 本章中最初定义的 **Date struct** 有什么局限和问题？
4. 为什么要为 **Date** 类型定义构造函数来取代函数 `init_day()`？
5. 什么是不变式？给出一个例子。
6. 什么时候应该将函数定义置于类定义内？什么时候又应该置于类外？为什么？
7. 在程序中什么时候应该使用运算符重载？给出一个你可能想重载的运算符列表（对于每一个请给出一个原因）。
8. 为什么应该令一个类的公有接口尽量小？
9. 为一个成员函数加上 `const` 限定符有什么作用？
10. 为什么辅助函数最好放在类定义之外？

术语

<code>built-in types</code> (内置类型)	<code>in-class initializer</code> (类内初始化值)
<code>class</code>	<code>inlining</code> (内联)
<code>const</code>	<code>interface</code> (接口)
<code>constructor</code> (构造函数)	<code>invariant</code> (不变式)
<code>destructor</code> (析构函数)	<code>representation</code> (表示)
<code>enum</code>	<code>struct</code>
<code>enumeration</code> (枚举)	<code>structure</code> (结构)
<code>enumerator</code> (枚举量)	<code>user-defined types</code> (用户自定义类型)
<code>helper function</code> (辅助函数)	<code>valid state</code> (合法状态)
<code>implementation</code> (实现)	

习题

1. 对 9.1 节中介绍的真实世界中的对象（如烤面包机）列出可能的操作。
2. 设计并实现一个保存（名字，年龄）对的 `Name_pairs` 类，其中名字是一个 `string`，年龄是一个 `double`。将值对表示为一个名为 `name` 的 `vector<string>` 成员和一个名为 `age` 的 `vector<double>` 成员。提供一个输入操作 `read_name()`，能读入一个名字列表。提供一个 `read_ages()` 操作，提示用户为每个名字输入一个年龄。提供一个 `print()` 操作，按 `name` 向量的顺序打印 (`name[i], age[i]`) 对（每行一个值对）。提供一个 `sort()` 操作，将 `name` 向量按字典序排序，并重整 `age` 向量与 `name` 向量新顺序匹配。将所有“操作”实现为成员函数。测试这个类（当然，在设计过程中尽早测试并多测试）。
3. 将 `Name_pairs::print()` 函数替换为（全局）运算符 `<<`，并为 `Name_pairs` 定义 `==` 和 `!=` 运算符。

4. 考察 8.4 节最后那个令人头痛的例子。给它加上适当的缩进和解释每个语法结构意义的注释。注意，这个例子并未做任何有意义的事情，它只是单纯为了说明令人困惑的代码风格。
5. 此题和后面几题要求你设计并实现一个 **Book** 类，你可以设想这是图书馆软件系统的一部分。**Book** 类应包含表示 ISBN 号、书名、作者和版权日期的成员，以及表示是否已经借出的成员。创建能返回这些成员的值的函数，以及借书和还书的函数。对于输入 **Book** 对象的数据进行简单的合法性检查，例如：只接受 **n-n-n-x** 形式的 ISBN 号，其中 **n** 是一个整数，**x** 是一个数字或一个字母。将 ISBN 号存储为 **string**。
6. 为 **Book** 类添加运算符。添加 **==** 运算符，用于检查两本书的 ISBN 号是否相等。定义 **!=** 运算符，比较 ISBN 号是否不等。定义 **<<**，分行输出书名、作者和 ISBN 号。
7. 为 **Book** 类创建一个名为 **Genre** 的枚举类型，用以区分书籍的类型：小说 (**fiction**)、非小说类文学作品 (**nonfiction**)、期刊 (**periodical**)、传记 (**biography**)、儿童读物 (**children**)。为每本书赋予一个 **Genre** 值，适当修改 **Book** 的构造函数和成员函数。
8. 为图书馆创建一个 **Patron** 类，包含读者姓名、图书证号及逾期费 (如果欠费的话)。创建访问这些成员的函数和设定逾期费的函数。定义一个辅助函数，返回一个布尔值，表示读者是否欠费。
9. 创建一个 **Library** 类，包含一个 **Book** 向量和一个 **Patron** 向量。定义一个名为 **Transaction** 的 **struct**，包含一个 **Book** 对象、一个 **Patron** 对象和一个本章中定义的 **Date** 对象，表示借阅记录。在 **Library** 类中定义一个 **Transaction** 向量。定义向图书馆添加图书、添加读者以及借出书籍的函数。当一个读者借出一本书时，保证 **Library** 对象中有此读者和这本书的记录，否则报告错误。然后检查读者是否欠费，如果欠费就报告一个错误，否则创建一个 **Transaction** 对象，将其放入 **Transaction** 向量中。定义一个返回包含所有欠费读者姓名的向量的函数。
10. 实现 9.8 节中的 **leapyear()**。
11. 为 **Date** 类设计并实现一组辅助函数，如 **next_workday** (假定除周六和周日外都是工作日) 和 **week_of_year** (假定第 1 周是 1 月 1 日所在那周，每周的第 1 天是周日)。
12. 改变 **Date** 类的描述，用 1970 年 1 月 1 日 (第 0 天) 至今的天数表示日期，用一个 **long int** 型成员保存此天数，重新实现 9.8 节中的函数。确保拒绝用这种方法无法表示的日期 (第 0 天之前的日期也拒绝，即不允许负数天数)。
13. 设计并实现一个有理数类 **Rational**。一个有理数由两部分组成——分子和分母，如 **5/6** (六分之五，或近似为 0.833 33)。如果需要的话，请查找有理数的定义。为 **Rational** 类定义实现赋值、加、减、乘、除及相等判定的运算符，并定义转换至 **double** 型值的函数。为什么人们需要使用 **Rational** 类？
14. 设计并实现一个 **Money** 类，能进行包含美元和美分的计算，精确到美分，使用四舍五入规则 (大于等于 0.5 美分入，小于 0.5 美分舍)。用一个 **long int** 型成员以美分值表示金额，但输入输出采用美元和美分的形式，如 \$123.45。不用考虑金额值超出 **long int** 型范围的情况。
15. 改进 **Money** 类，加入货币功能 (货币类型通过构造函数参数给出)。能接受浮点型的初值，只要能用 **long int** 型准确表示即可。不允许非法操作，如 **Money*Money** 这种无意义的操作，但 **USD1.23+DKK5.00** 这种有意义的操作，只要你提供了美元 (USD) 和丹麦克

朗 (DKK) 之间的汇率就可以支持。

16. 定义输入运算符 (`>>`)，可读入货币数量和名称，如 `USD1.23` 和 `DKK5.00`，存储到一个 `Money` 变量中。再定义相应的输出运算符 (`<<`)。
17. 给出一个例子，使用 `Rational` 进行计算得到的结果比使用 `Money` 更好。
18. 给出一个例子，使用 `Rational` 进行计算得到的结果比使用 `double` 类型更好。

附言

用户自定义类型是非常多的，比本章所介绍的多得多。用户自定义类型，特别是类，是 C++ 的核心，是很多高效设计技术的关键。在本书剩余部分中，大部分内容都是关于类的设计和使用的。一个类，或者一组类，是我们用来在代码中表达思想的机制。本章主要介绍了类的语言技术细节，本书其他部分则关注如何用类优美地表达有用的思想。

输入输出流

我们学习科学来远离愚昧。

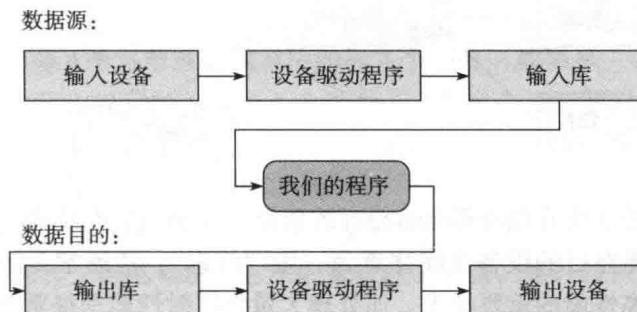
——Richard P. Feynman

在本章和下一章中，我们将从多个角度来学习 C++ 标准库中有关处理输入输出的特性：输入输出流。我们将展示如何读写文件，如何处理输入输出错误，如何进行格式化输入，以及如何为用户自定义类型提供输入输出运算符及如何使用这类操作符。本章重点介绍基本问题：如何读写单个值，以及如何打开 / 读 / 写整个文件。最后会用一个例子来说明在一个较大规模的程序中应该考虑的一些 I/O 相关问题。下一章会介绍更深入的技术细节。

10.1 输入和输出

如果没有数据，计算就毫无意义。我们需要将数据输入到程序中来进行一些有意义的计算，并将结果从程序中取出。在 4.1 节中我们曾经提及，数据的输入源和输出目标多种多样令人眼花缭乱。如果我们不注意输入源和输出目标的处理，就会写出只能从特定的源输入数据，将结果输出到特定设备的程序。这对于某些特定应用，比如数码相机或用于引擎燃料喷射器的传感器来说，是可以接受的（有时甚至是必需的）。但对于大多数应用，我们需要某种方法将程序的读写操作与实际进行输入输出的设备分离开。如果必须直接访问每种设备，那么当有新的显示器或磁盘产品面市时，我们就必须修改程序，或者将用户局限于程序所支持的设备，这是很荒谬的。

大多数现代操作系统都将 I/O 设备的处理细节放在设备驱动程序中，通过一个 I/O 库访问设备驱动程序，这就使不同设备源的输入输出尽可能地相似。一般地，设备驱动程序都位于操作系统较深的层次中，大多数用户是看不到它们的。I/O 库给出了输入输出的一个抽象，从而令程序员不必关心具体的设备和设备驱动程序：



如果操作系统使用这样一个模型，则所有输入和输出都可以看作字节（字符）流，由输入输出库处理。更多复杂的 I/O 模式需要更专门的知识，已超出了本书的范围。因此，我们程序员的工作就变为：

1. 创建指向恰当数据源和数据目的的 I/O 流。

2. 从这些流中读取数据或将数据写入到这些流中。

数据在程序和设备间实际是如何传输的呢？这类细节都是由 I/O 库和驱动程序来处理的。

在本章和下一章中，我们将介绍如何使用 C++ 标准库来处理包含了格式化数据的 I/O 流。

 从程序员的角度来看，输入和输出可以分为多种不同类型：

- (大量) 数据项构成的流 (通常对应文件、网络连接、录音设备或显示设备)。
- 通过键盘来与用户交互的流。
- 通过图形界面与用户交互的流 (输出对象、接收鼠标点击事件，等等)。

此分类法并非唯一可能的分类，而且在这种分类中，三类 I/O 流的划分不是那么清晰。

例如，如果一个输出字符流是一个以浏览器为目的地的 HTTP 文档，那么它看起来更像一个与用户交互的流，而且它可以包含图形元素。相反，与 GUI (用户图形界面) 交互的流也可能以一个字符序列的形式呈现给程序。然而，这种分类很适合我们的工具：前两类 I/O 可以用 C++ 标准库中的 I/O 流实现，而且大多数操作系统都直接支持这两类 I/O。从第 1 章开始，我们就已经使用 `iostream` 库了，在本章和下一章中，我们仍主要关注这方面的内容。图形化输出和图形化用户交互则由其他一些库支持，我们将在第 17 章至第 21 章中讨论这部分内容。

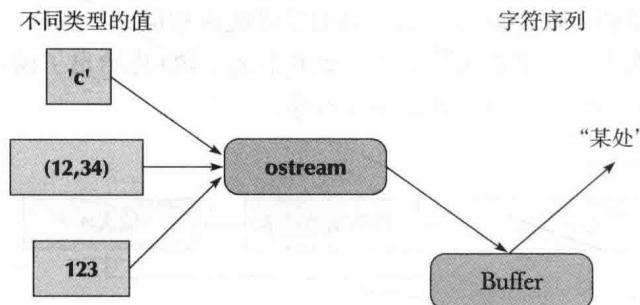
10.2 I/O 流模型

C++ 标准库提供了两种数据类型，`istream` 用于处理输入流，`ostream` 用于处理输出流。我们已经使用过标准输入流 `cin` 和标准输出流 `cout`，因此我们已经了解了应该如何使用标准库中的这部分特性（通常称之为 `iostream` 库）。

 一个 `ostream` 可以实现：

- 将不同类型的值转换为字符序列。
- 将这些字符发送到“某处”(如控制台、文件、主存或者另外一台计算机)。

我们可以用下图来表示 `ostream`：

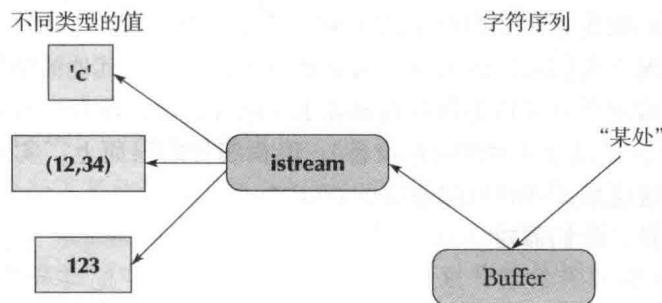


`Buffer` 这一数据结构用于保存提交给 `ostream` 的数据，并通过它与操作系统通信。如果在写入 `ostream` 和字符出现在目的设备之间注意到一段“延迟”，这通常是因为字符还在缓冲区之中。缓冲技术是提高性能的重要技术，而处理大量数据时性能是很重要的。

 一个 `istream` 可以实现：

- 将字符序列转换为不同类型的值。
- 从某处 (如控制台、文件、主存或另外一台计算机) 获取字符。

我们可以用下图来表示 `istream`：



与 `ostream`一样，`istream` 也使用一个缓冲区与操作系统通信。`istream` 的缓冲区在很多情况下对用户是可见的。当使用一个与键盘相关的 `istream` 时，键入的内容都被留在缓冲区中，直至按回车键为止（回车 / 换行），也可以使用清除键（退格）来“改变你的主意”（直至按回车键为止）。

输出的一个主要目的就是生成可供人们阅读的数据形式，例如 `email` 消息、学术论文、网页、账单记录、商务报告、通讯录、目录、设备状态信息等实例。因此，`ostream` 提供了很多特性，用于格式化文本以适应不同需求。同样，为了易于人们阅读，很多输入数据也是由人们事先编写或者格式化过的。因此，`istream` 提供了一些特性，用于读取由 `ostream` 生成的输出内容。我们将在 11.2 节介绍格式化输入输出，在 11.3.2 节中介绍如何读取非字符型输入数据。大多数和输入相关的复杂性，都与如何进行错误处理有关系。为了能给出更为实际的例子，我们将从 `iostream` 模型如何与数据文件相关联开始讨论。

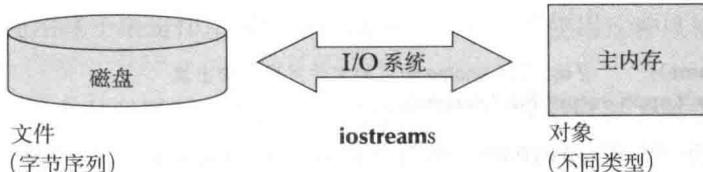
10.3 文件

通常，我们需要处理的数据会多得难以放入计算机的内存之中，因此我们将大部分数据存放于磁盘或其他大容量存储设备中。这种设备还具有另外一个我们需要的特性：断电后，保存在其中的数据不会丢失——即数据是持久的。究其根本，一个文件可以简单看作一个从 0 开始编号的字节序列。



每个文件都有自己的格式，也就是说，有一组规则来确定其中字节的含义。例如，如果是一个文本文件，前 4 个字节就是内容中的前 4 个字符。如果是一个二进制表示的整数文件，同样是前 4 个字节，表示的就是第 1 个整数了（参见 11.3.2 节）。格式之于磁盘文件的作用，与类型之于内存对象的作用是一样的。当（且仅当）我们知道一个文件的格式（参见 11.2.3 节），我们就能弄清文件中比特流的含义了。

对于一个文件，`ostream` 将内存中的对象转换为字节流，再将字节流写到磁盘上。`istream` 进行相反的操作，也就是说，它从磁盘获取字节流，将其转换为对象。



多数情况下，我们假定这些“磁盘上的字节”都是常用字符集中的字符。事实并不总是如此，但绝大多数情况下我们可以认为这个假定是对的，而且，其他的字符集表示方式也是不难处理的。我们还假定所有文件都保存在磁盘上（也就是说，保存在旋转磁存储设备上）。同样，这个假设也不总是成立（考虑闪存设备），但在编程的层面上，实际采用哪种存储设备是没什么区别的，这也是文件和流抽象层的好处之一。

为了读取一个文件，我们需要：

1. 知道文件名。
2. (以读模式) 打开文件。
3. 读出字符。
4. 关闭文件 (虽然通常文件会被隐式地关闭)。

为了写一个文件，我们需要：

1. 指定文件名。
2. 按照指定的文件名，(以写模式) 打开文件或创建一个新文件。
3. 写入我们的对象。
4. 关闭文件 (虽然通常文件会被隐式地关闭)。

实际上我们已经掌握了基本的文件读写方法了，因为关联到一个文件的 **ostream** 对象的使用方式与 **cout** 是完全一样的，**istream** 对象则与 **cin** 完全一样，而对于 **cin** 和 **cout**，我们已经很熟悉了。我们会在 11.3.3 节中介绍一些只用于文件的操作。但现在，我们还是先了解如何打开文件，关注那些可以用于所有 **ostream** 和 **istream** 对象的操作和技术。

10.4 打开文件

如果要读或写一个文件，需要打开一个与文件相关联的流。**ifstream** 是用于读取文件的 **istream** 流，**ofstream** 是用于写文件的 **ostream** 流，**fstream** 是用于对文件进行读写的 **iostream** 流，文件流必须与某个文件相关联，然后才可使用。例如：

```
cout << "Please enter input file name: ";
string iname;
cin >> iname;
ifstream ist {iname};      // ist 是以 iname 命名的文件对应的输入流
if (!ist) error("can't open input file ", iname);
```

用一个名字字符串定义一个 **ifstream**，可以打开以该字符串为名的文件进行读操作。**!ist** 检测文件是否成功打开。如果成功打开，我们可以像处理其他任何 **istream** 那样从文件中读取数据。例如，假定已经对 **Point** 类定义了输入运算符 **>>**，可以写出如下的代码：

```
vector<Point> points;
for (Point p; ist>>p; )
    points.push_back(p);
```

写文件的过程与读文件类似，通过流 **ofstream** 来实现，例如：

```
cout << "Please enter name of output file: ";
string oname;
cin >> oname;
ofstream ost {oname};      // ost 是以 oname 命名的文件对应的输出流
if (!ost) error("can't open output file ", oname);
```

用一个名字字符串定义一个 **ofstream**，会打开以该字符串为名的文件与流相关联。**!ost** 检测

文件是否成功打开。如果打开成功，我们就可以像处理其他 `ostream` 对象一样向文件中写入数据，例如：

```
for (int p : points)
    ost << '(' << p.x << ',' << p.y << ")\n";
```

当一个文件流离开了其作用域，与之关联的文件就会被关闭。当文件被关闭时，与之关联的缓冲区会被刷新，也就是说，缓冲区中的字符会被写入到文件中。

一般来说，最好在程序中一开始的位置，在任何重要的计算都尚未开始之前就打开文件。毕竟，如果我们在完成计算之后才发现无法保存结果，将会是对计算资源的巨大浪费。

理想的方法是在创建 `ostream` 或 `istream` 对象时隐式打开文件，并依靠流对象的作用域来关闭文件。例如：

```
void fill_from_file(vector<Point>& points, string& name)
{
    ifstream ist {name};           // 打开文件准备读
    if (!ist) error("can't open input file ", name);
    // 使用 ist
    // 在退出函数时文件被隐式关闭
}
```

此外，还可以通过 `open()` 和 `close()` 操作显式打开和关闭文件（参见附录 C.7.1）。但是，依靠作用域的方式最大程度地降低了两类错误出现的概率：在打开文件之前或关闭之后使用文件流对象。例如：

```
ifstream ifs;
// ...
ifs >> foo;                      // 不会成功：没有为 ifs 打开的文件
// ...
ifs.open(name,ios_base::in);       // 打开以 name 命名的文件进行读操作
// ...
ifs.close();                      // 关闭文件
// ...
ifs >> bar;                      // 不会成功：ifs 对应的文件已经关闭
// ...
```

在真实的程序中，通常这类错误更难以定位。幸运的是，我们不能在还没有关闭一个文件时就第二次打开它。例如：

```
fstream fs;
fs.open("foo", ios_base::in);      // 打开文件进行读操作
// 缺失了 close()
fs.open("foo", ios_base::out);     // fs 对应的文件已经打开
if (!fs) error("impossible");
```

因此，在打开一个文件之后，一定不要忘记检测流对象是否成功关联了。

那么为什么还要显式使用 `open()` 和 `close()` 操作呢？原因是偶尔会遇到这样的情况——使用文件的范围不能简单包含于任何流对象的作用域中，此时我们就不得不这样做了。但这种情况非常少见，所以我们不必为此担心。更确切地说，`iostream`（以及 C++ 标准库的其他部分）都使用基于限定作用域的编程风格，只有在不使用这种风格的代码中才会遇到上述情况。

在第 11 章中我们会看到更多文件相关的话题，但现在我们只要了解它们作为数据源和数据目的的使用方法就够了。如果假定用户必须直接键入所有输入数据，那么写出的程序会

非常不实用。从程序员的角度来看，使用文件的优点在于可以在调试过程中反复从文件读取输入，直到程序运行正确。

10.5 读写文件

思考这样一个问题：你如何从一个文件中读取一些测量实验结果，并在内存中将它们呈现出来？这些实验结果可能是从气象站获取的温度数据：

```
0 60.7
1 60.6
2 60.3
3 59.22
...
...
```

这个数据文件由一个（小时，温度）数值对序列组成。小时的值从 0 到 23，温度为华氏度值。假定文件没有更多的格式，也就是说，这个文件不包含任何特殊的头信息（例如温度读数是从哪里获取的）、值的单位、标点（例如为每对数值加上括号）或者终止符。这是一个最为简单的情形。

我们可以用一个 **Reading** 类型来描述温度读数：

```
struct Reading {           // 温度数据读取
    int hour;              // 在 [0:23] 区间取值的小时数
    double temperature;    // 华氏度值
};
```

有了这样的类型，我们可以按如下方式来读取温度读数：

```
vector<Reading> temps;      // 在这里存储读数
int hour;
double temperature;
while (ist >> hour >> temperature) {
    if (hour < 0 || 23 < hour) error("hour out of range");
    temps.push_back(Reading{hour,temperature});
}
```

这是一个典型的输入循环。如上一节所示，**istream** 流 **ist** 可以是一个输入文件流 (**ifstream**)，也可以是标准输入流 **cin**(的一个别名)，或者是任何其他类型的 **istream**。对于这段代码而言，它并不关心这个 **istream** 是从哪里获取数据。我们的程序所关心的只是：**ist** 是一个 **istream**，而且数据格式如我们所期望。下一节我们将讨论一个有趣的问题：如何检测输入数据中的错误，以及发现格式错误后该如何处理。

写文件通常比读文件要简单。再重复一遍，一旦一个流对象已经被初始化，我们就可以不必了解它到底是哪种类型的流。特别地，对于上一节介绍的输出文件流 (**ofstream**)，我们可以像使用其他任何 **ostream** 一样来使用它。例如，我们可能想输出带括号的温度读数数值对：

```
for (int i=0; i<temps.size(); ++i)
    ost << '(' << temps[i].hour << ',' << temps[i].temperature << ")\n";
```

最终的程序结果是，读取原始的温度读数文件，然后产生一个新文件，其中的数据格式为（小时，温度）。

由于文件流对象在离开其作用域时会自动关闭所关联的文件，因此完整的程序如下所示：

```

#include "std_lib_facilities.h"

struct Reading {
    int hour;           // 温度数据读取
    double temperature; // 在 [0:23] 区间取值的小时数
};                   // 华氏度值

int main()
{
    cout << "Please enter input file name: ";
    string iname;
    cin >> iname;
    ifstream ist {iname};      // ist 读取以 iname 命名的文件
    if (!ist) error("can't open input file ", iname);

    string oname;
    cout << "Please enter name of output file: ";
    cin >> oname;
    ofstream ost {oname};       // ost 写入以 oname 命名的文件
    if (!ost) error("can't open output file ", oname);

    vector<Reading> temps;    // 在这里存储读数
    int hour;
    double temperature;
    while (ist >> hour >> temperature) {
        if (hour < 0 || 23 < hour) error("hour out of range");
        temps.push_back(Reading{hour, temperature});
    }

    for (int i=0; i<temps.size(); ++i)
        ost << '(' << temps[i].hour << ','
            << temps[i].temperature << ")\n";
}

```

10.6 I/O 错误处理

当处理输入时，我们必须预计到其中可能发生的错误并给出相应的处理措施。输入中会发生什么类型的错误呢？应该如何处理呢？输入错误可能是由于人的失误（错误理解了指令、打字错误、允许自家的小猫在键盘上散步等）、文件格式不符、我们（程序员）错误估计了情况等等原因造成的。发生输入错误的可能情况是无限的！但 `istream` 将所有可能的情况归结为四类，称为流状态（stream state）：

流状态	
<code>good()</code>	操作成功
<code>eof()</code>	到达输入末尾（“文件尾”）
<code>fail()</code>	发生某些意外情况（例如，我们要读入一个数字，却读入了字符‘x’）
<code>bad()</code>	发生严重的意外（如磁盘读故障）

不幸的是，`fail()` 和 `bad()` 之间的区别并未被准确定义，（定义新类型 I/O 操作的）程序员对此的观点各种各样。但是，基本的思想很简单：如果输入操作遇到一个简单的格式错误，则使流进入 `fail()` 状态，也就是假定我们（输入操作的用户）可以从错误中恢复。另一方面，如果错误真的非常严重，例如发生了磁盘读故障，输入操作会使得流进入 `bad()` 状态。

态，也就是假定面对这种情况你所能做的很有限，只能退出输入。这种观点导致如下逻辑：

```

int i = 0;
cin >> i;
if (!cin) { // 只有输入操作失败，才会跳转到这里
    if (cin.bad()) error("cin is bad"); // 流发生故障：让我们跳出程序！
    if (cin.eof()) {
        // 没有任何输入
        // 这是我们结束程序经常需要的输入操作序列
    }
    if (cin.fail()) { // 流遇到了一些意外情况
        cin.clear(); // 为更多的输入操作做准备
        // 恢复流的其他操作
    }
}

```

`!cin` 可以理解为“`cin` 不成功”或者“`cin` 发生了某些错误”或者“`cin` 的状态不是 `good()`”，这与“操作成功”正好相反。请注意我们在处理 `fail()` 时所使用的 `cin.clear()`。当流发生错误时，我们可以进行错误恢复。为了恢复错误，我们显式地将流从 `fail()` 状态转移到其他状态，从而可以继续从中读取字符。`clear()` 就起到这样的作用——执行 `cin.clear()` 后，`cin` 的状态就变为 `good()`。

下面是一个如何使用流状态的例子。假定我们要读取一个整数序列存入 `vector` 中，字符“*”或“文件尾”(在 Windows 平台是字符 `Ctrl+Z`，Unix 平台是 `Ctrl+D`) 表示序列结束。例如：

1 2 3 4 5 *

上述功能可通过如下函数来实现：

```

void fill_vector(istream& ist, vector<int>& v, char terminator)
    // 从 ist 中读入整数到 v 中，直到遇到 eof() 或终结符
{
    for (int i; ist >> i; ) v.push_back(i);
    if (ist.eof()) return; // 发现到了文件尾

    if (ist.bad()) error("ist is bad"); // 流发生故障：让我们跳出程序！
    if (ist.fail()) { // 最好清除混乱，然后汇报问题
        ist.clear(); // 清除流状态
        // 以便寻找终结符
        char c;
        ist >> c; // 读入一个符号，希望是终结符
        if (c != terminator) { // 非终结符
            ist.unget(); // 放回该符号
            ist.clear(ios_base::failbit); // 将流状态设置为 fail()
        }
    }
}

```

注意，即使没有遇到终结符，函数也会返回。毕竟，我们可能已经读取了一些数据，而 `fill_vector()` 的调用者也许有能力从 `fail()` 状态中恢复过来。由于我们已经清除了状态以便检查后续字符，所以必须将流状态重新置为 `fail()`。我们通过调用 `ist.clear(ios_base::failbit)` 来达到这一目的。对照简单的 `clear()`，带参数的用法有些令人迷惑：当 `clear()` 调用带参数时，参数中所指出的 `iostream` 状态位会被置位（进入相应状态），而未指出的状态位会被复位。通过将流状态设置为 `fail()`，我们表明遇到了一个格式错误，而不是一个更为严重的问题。

可以用 `unget()` 将字符放回 `ist`, 以便 `fill_vector()` 的调用者可能使用该字符。`unget()` 函数是 `putback()` 的简化版本 (参见 6.8.2 节, 附录 C.7.3), 它依赖于流对象记住最后一个字符是什么, 所以在这里可以不用考虑它的用法。

如果调用了 `fill_vector()` 并且想知道是什么原因终止了输入, 那么可以检测流是处于 `fail()` 还是 `eof()` 状态。当然也可以捕获 `error()` 抛出的 `runtime_error` 异常, 但当 `istream` 处于 `bad()` 状态时, 继续获取数据是不可能的。大多数的调用者无须为此烦恼。因为这意味着, 几乎在所有情况下, 对于 `bad()` 状态, 我们所能做的只是抛出一个异常。简单起见, 可以让 `istream` 帮我们来做。

```
// 当 ist 出现问题时抛出异常  
ist.exceptions(ist.exceptions() | ios_base::badbit);
```

这样的写法也许看起来有些奇怪, 但结果却很简单, 当此语句执行时, 如果 `ist` 处于 `bad()` 状态, 它会抛出一个标准库异常 `ios_base::failure`。在一个程序中, 我们只需要调用 `exceptions()` 一次。这允许我们简化关联于 `ist` 的所有输入过程, 同时忽略对 `bad()` 的处理:

```
void fill_vector(istream& ist, vector<int>& v, char terminator)  
    // 从 ist 中读入整数到 v 中, 直到遇到 eof() 或终结符  
{  
    for (int i; ist >> i; ) v.push_back(i);  
    if (ist.eof()) return; // 发现到了文件尾  
  
    // 不是 good(), 不是 bad(), 不是 eof(), ist 的状态一定是 fail()  
    ist.clear(); // 清除流状态  
  
    char c;  
    ist >> c; // 读入一个符号, 希望是终结符  
  
    if (c != terminator) { // 不是终结符号, 一定是失败了  
        ist.unget(); // 也许程序调用者可以使用这个符号  
        ist.clear(ios_base::failbit); // 将流状态设置为 fail()  
    }  
}
```

这里使用了 `ios_base`, 它是 `iostream` 的一部分, 包含了对常量如 `badbit` 的定义、异常如 `failure` 的定义, 以及其他一些有用的定义。可以通过 `::` 操作符来使用它们, 例如 `ios_base::badbit` (参见附录 C.7.2)。我们无须如此深入地讨论 `iostream` 库的细节, 若要学习 `iostream` 的所有内容, 可能需要一门完整的课程。例如, `iostream` 可以处理不同的字符集, 实现不同的缓冲策略, 还包含一些工具, 能按不同语言的习惯格式化货币金额的输入输出。我们曾经收到过一份关于乌克兰货币输入输出格式的错误报告。如果需要了解更多 `iostream` 库的内容, 可以参考 Stroustrup 的《The C++ Programming Language》和 Langer 的《Standard C++ IOStreams and Locales》。

与 `istream` 一样, `ostream` 也有四个状态: `good()`、`fail()`、`eof()` 和 `bad()`。不过, 对于本书中的程序来说, 输出错误要比输入错误少得多, 因此通常不对 `ostream` 进行状态检测。如果程序运行环境中输出设备不可用、队列满或者发生故障的概率很高, 我们就可以像处理输入操作那样, 在每次输出操作之后都检测其状态。

10.7 读取单个值

现在我们已经知道如何读取以文件尾或者某个特定终结符结束的值序列了。我们还会学

习一些更为复杂的例子，但在此之前，先来看一个非常常见的应用问题：不断要求用户输入一个值，直至用户输入的值合乎要求为止。这个例子允许我们检验几种常见的设计策略，我们通过“如何从用户获取一个合乎要求的值”这个问题的几种可选的解决方案来讨论这些设计策略。我们从一个不那么令人满意的“初步尝试”方案开始，逐步提出改进方案。基本假设是：我们是在处理交互式的输入——程序给出提示信息，用户键入数据。假定我们要求用户输入一个 1 到 10 之间（包含 1 和 10）的整数：

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin>>n) {                                // 读操作
    if (1<=n && n<=10) break; // 检查范围
    cout << "Sorry "
    << n << " is not in the [1:10] range; please try again\n";
}
// 这里使用 n
```

这段代码确实很“丑陋”，但它在某种程度上是能正常运行的。如果你不喜欢使用 `break`（参见附录 A.6），可以将读操作和范围检查合并为一条语句：

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin>>n && !(1<=n && n<=10)) // 读操作，同时检查范围
    cout << "Sorry "
    << n << " is not in the [1:10] range; please try again\n";
// 这里使用 n
```

⚠ 这不过是简单的改头换面，并未改变代码的实质。为什么说这段代码只是“某种程度上能正常运行”的呢？原因在于，这段代码只有在用户很小心地输入整数的情况下才正常运行。如果用户用键盘输入很不熟练，本想输入 6，但输入了 t（在大多数键盘上，t 恰好在 6 的下面），程序会不改变 n 的值就退出循环，于是 n 中的值就不在要求范围之内。这样的代码是不能被称为高质量代码的。而且，爱开玩笑的人（或者是勤奋的测试人员）还有可能从键盘键入“文件尾”符号（在 Windows 平台是 Ctrl+Z，在 Unix 平台是 Ctrl+D）。于是，再次出现循环结束后 n 不在合法范围的情况。换句话说，为了获得可靠的输入，我们必须处理三个问题：

1. 用户输入超出范围的值。
2. 没有输入任何值（输入文件尾符号）。
3. 用户输入的内容类型错误（本例中，未输入整数）。

我们要如何应对这些问题呢？这也是程序设计过程中常常会遇到的问题：我们真正想要的是什么？在这里，对于上述每个错误，我们有三种可选的应对方式：

1. 在负责输入的代码中处理错误。
2. 抛出一个异常，让其他代码来处理这个错误（有可能终止程序）。
3. 忽略这个错误。

巧合的是，这恰恰是三种最为常用的错误处理策略，因此本例非常适合展示如何处理错误。

表面来看，第三种策略（即忽略错误的方式）是不可接受的，但这样说有点过于武断。如果我是在编写一个自己用的简单程序，还是可以随意选择实现策略的，包括“忘记”进行错误检测而可能产生糟糕结果。但是，如果我是在编写一个将来可能长时间运行的程序，忽略这些错误就可能很愚蠢了。如果程序可能被他人所用的话，就更加不能在程序中忽略对这

类错误的检测了。请注意，这里有意识地使用了第一人称“我”，因为“我们”可能会导致误解。也就是说，我们的观点是，即使只有两个人使用程序，第三种策略也是不可接受的。

在第一和第二种策略间进行取舍是很困难的。对于某个给定程序，可能有很好的理由选择两种策略中的任何一个。首先要注意，在大多数程序中，对于用户没有通过键盘输入任何数据的情况，还没有一种简洁的、局部性的方法来处理：因为当输入流关闭后，没有其他好的办法来请求用户输入一个数。我们当然可以重新打开 cin（使用 cin.clear()），但用户很可能不是意外地关闭输入流的（你会意外地敲 Ctrl+Z 键吗？）。如果一个程序希望读取一个整数，但却遇到一个“文件尾”，负责读入整数的代码必须放弃努力并寄希望于程序的其他部分能处理这个问题，也就是说，读取输入的代码应该抛出一个异常。这意味着我们并不是要选择是抛出异常还是就地处理错误，而是要选择哪些错误应就地处理。

10.7.1 将程序分解为易管理的子模块

下面我们尝试既处理超出范围的输入，又处理类型错误的输入：

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (true) {
    cin >> n;
    if (cin) {           // 获得一个整数，现在检查该整数
        if (1<=n && n<=10) break;
        cout << "Sorry "
            << n << " is not in the [1:10] range; please try again\n";
    }
    else if (cin.fail()) {          // 发现了非整数的符号
        cin.clear();               // 将状态设置为 good()
        // 我们想要查看这些符号
        cout << "Sorry, that was not a number; please try again\n";
        for (char ch; cin>>ch && !isdigit(ch); ) // 忽略非数值符号
            // 什么也不做
        if (!cin) error("no input"); // 没有发现数字，放弃
        cin.unget(); // 将数字放回，这样就可以读出一个数
    }
    else {
        error("no input");         // eof 或者 bad 状态：放弃
    }
}
// 如果这里我们得到了 [1:10] 中的 n
```

这段代码又乱又冗长。当有人需要编写让用户输入整数的程序时，我们绝不会建议他们这样写。但另一方面，我们确实要在代码中处理潜在的错误，因为用户确实制造了错误，我们该怎么办呢？这段程序如此之乱，是因为它把处理好几件不同事情的代码都混合在一起了：

- 读取数值。
- 提示用户输入。
- 输出错误信息。
- 跳过“问题输入字符”。
- 测试输入是否在所需范围内。

一种常用的令代码更为清晰的方法是将逻辑上做不同事情的代码划分为独立的函数。例

如，对于发现“问题字符”（如意料之外的字符）后进行错误恢复的代码，就可以将其分离出来：

```
void skip_to_int()
{
    if (cin.fail()) {           // 我们发现了非整数的符号
        cin.clear();           // 我们想要查看这些符号
        for (char ch; cin>>ch;) { // 忽略非数值符号
            if (isdigit(ch) || ch=='-') {
                cin.unget();      // 将数字放回
                // 这样就可以读出一个数
                return;
            }
        }
    }
    error("no input");         // eof 或者 bad 状态：放弃
}
```

有了上面的“工具函数”`skip_to_int()`后，代码就可以改写为：

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (true) {
    if (cin>>n) {           // 获得一个整数，现在检查该整数
        if (1<=n && n<=10) break;
        cout << "Sorry " << n
        << " is not in the [1:10] range; please try again\n";
    }
    else {
        cout << "Sorry, that was not a number; please try again\n";
        skip_to_int();
    }
}
// 如果这里我们得到了 [1:10] 中的 n
```

这段代码就好多了，但它还是太长、太乱，很难在程序中多次使用。我们需要进行大量的测试，才能保证其正确性。

我们到底需要什么样的操作呢？一个看起来挺合理的答案是：“我们需要一个读取任意整数的函数，以及一个读取指定范围内整数的函数。”

```
int get_int();                  // 从 cin 中读取一个整数
int get_int(int low, int high); // 从 cin 中读取介于 [low:high] 的整数
```

如果有这些函数，我们至少能简单而又正确地使用它们。不难写出：

```
int get_int()
{
    int n = 0;
    while (true) {
        if (cin >> n) return n;
        cout << "Sorry, that was not a number; please try again\n";
        skip_to_int();
    }
}
```

`get_int()`持续读入字符，直至发现可以解释为整数的数字符号为止。如果想让`get_int()`结束，必须输入一个整数或者文件尾符号（文件尾符号会使`get_int()`抛出一个异常）。

使用通用版本的`get_int()`，可以写出具有范围检查功能的`get_int()`：

```

int get_int(int low, int high)
{
    cout << "Please enter an integer in the range "
        << low << " to " << high << " (inclusive):\n";

    while (true) {
        int n = get_int();
        if (low <= n && n <= high) return n;
        cout << "Sorry "
            << n << " is not in the [" << low << ':' << high
            << "] range; please try again\n";
    }
}

```

这个版本的 `get_int()` 同样很固执，它利用普通 `get_int()` 不断读取整数，直至读入的值在所需范围之内。

现在可以使用下面的代码读取整数：

```

int n = get_int(1,10);
cout << "n: " << n << '\n';

int m = get_int(2,300);
cout << "m: " << m << '\n';

```

但是不要忘记在程序的某处捕获异常，这样，当 `get_int()` 真的不能读入一个整数时（虽然可能很罕见），我们就可以给出恰当的错误信息。

10.7.2 将人机对话从函数中分离

现在的 `get_int()` 函数中还是混合着读取输入的代码和输出提示信息的代码。对于一个简单程序来说，这可能没有什么问题。但在一个大型程序中，我们可能想要对用户输出不同的提示信息。例如，我们可能想要这样来调用 `get_int()`：

```

int strength = get_int(1,10, "enter strength", "Not in range, try again");
cout << "strength: " << strength << '\n';

int altitude = get_int(0,50000,
    "Please enter altitude in feet",
    "Not in range, please try again");
cout << "altitude: " << altitude << "f above sea level\n";

```

一种可能的实现如下：

```

int get_int(int low, int high, const string& greeting, const string& sorry)
{
    cout << greeting << ": [" << low << ':' << high << "]\n";

    while (true) {
        int n = get_int();
        if (low <= n && n <= high) return n;
        cout << sorry << ": [" << low << ':' << high << "]\n";
    }
}

```

生成任意提示信息是很困难的，所以我们采取了“固定风格”式的处理方式。这种处理方法可以生成任意可变的提示信息，比如支持很多自然语言（如阿拉伯语、孟加拉语、中文、丹麦语、英文以及法文），因此通常情况下是被人们所接受的。但对于初学者来说，这些都是超出学习范围的内容。

需要注意的是，我们的解决方案仍是不完整的：不进行范围检查的 `get_int()` 版本仍然会“信口胡言”。这里所体现出的深层次的问题是：“工具函数”会在程序中很多地方被调用，因此不应该将提示信息“硬编码”到函数中。更进一步，库函数会在很多程序中被使用，也不应该向用户输出任何信息——毕竟，编写库的程序员甚至可能不知道使用库函数的程序所运行的计算机是否有人在操作，因此在库函数中输出信息有可能毫无意义。这也是为什么我们的 `error()` 函数并没有输出任何错误信息（参见 5.6.3 节），因为一般来说，我们无法知道向何处输出。

10.8 用户自定义输出运算符

为一个给定类型定义输出运算符 `<<`，是一件很简单的事情。要考虑的主要问题是不同人可能喜欢不同的输出形式，因此很难达成共识来确定一个单一的格式。但即便无法提供一个令所有人都满意的单一输出格式，为用户自定义类型定义输出运算符 `<<`，通常也是一个好的策略。这样，我们至少可以在调试和早期开发期间，很容易地输出这个类型的对象。接下来，我们还可以提供一个更为复杂的 `<<`，允许用户给出格式信息。而且，如果我们希望输出样式与 `<<` 提供的不同，可以简单地绕过 `<<`，按照我们希望的格式直接输出用户自定义类型中的内容。

下面是为 9.8 节中 `Date` 类型定义的一个简单的输出运算符，它简单地打印年、月、日，中间用逗号分隔，两边加括号：

```
ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
        << ',' << d.month()
        << ',' << d.day() << ')';
}
```

这个输出运算符会将 2004 年 8 月 30 日打印为“(2004, 8, 30)”的形式。这种简单的成员列表的表示方式，对于成员数较少的类型来说很适合，除非我们有更好的想法或者更特殊的需求。

在 9.6 节中，我们提到，处理一个用户自定义运算符，实际是调用对应的函数。下面的例子演示了这一处理过程，假定已经为 `Date` 定义了上面的 `<<` 操作符，那么

```
cout << d1;
```

其中 `d1` 是 `Date` 类型的对象，等价于下面的调用：

```
operator<<(cout,d1);
```

需要注意 `operator<<()` 是如何接受一个 `ostream&` 作为第一个参数，又将其作为返回值返回的。这就是为什么可以将输出操作“链接”起来的原因，因为输出流按这种方式逐步传递下去了。例如，可以像下面这样输出两个日期：

```
cout << d1 << d2;
```

这里，将先处理第一个 `<<`，然后再处理第二个 `<<`。

```
cout << d1 << d2;      // 意味着 operator<<(cout, d1)<<d2;
                        // 意味着 operator<<(operator<<(cout, d1), d2);
```

也就是说，连续输出两个对象 `d1` 和 `d2`，`d1` 的输出流是 `cout`，而 `d2` 的输出流是第一个输出操作的返回结果。实际上，以上三种写法中任何一种都可以输出 `d1` 和 `d2`。当然，哪种最简洁、易读是一目了然的。

10.9 用户自定义输入运算符

为一个给定类型和指定的输入格式定义输入运算符 `<<`，关键在于错误处理。这可能会是很棘手的事情。

下面是为 9.8 节中 `Date` 类型定义的一个简单的输入运算符，它要求的输入格式与上一节定义的 `>>` 的输出格式相同：

```
istream& operator>>(istream& is, Date& dd)
{
    int y, m, d;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;
    if (ch1 != '(' || ch2 != ',' || ch3 != '=' || ch4 != ')') { // 格式错误
        is.clear(ios_base::failbit);
        return is;
    }
    dd = Date{y, Date::Month(m), d}; // 更新 dd
    return is;
}
```

`>>` 运算符读入形如 “(2004,8,20)” 的数据项，并尝试用这三个整数创建一个 `Date` 对象。和前面提到的一样，这里的输入处理要比输出处理难得多。输入比输出更容易出错，实际应用中看也确实如此。

如果未发现“(整数，整数，整数)”格式的输入，`>>` 运算符会令流进入一个非正常状态(`fail`、`eof` 或 `bad`)，并且不会改变目标 `Date` 对象的值。成员函数 `clear()` 用来设置流的状态。显然，`ios_base::failbit` 将使流进入 `fail()` 状态。理想的目标是在输入故障的情况下保持目标 `Date` 对象不变，而且这会使代码更干净。对于一个 `operator>>()` 来说，理想目标是不读取或丢弃任何它未用到的字符，但这太困难了：因为在捕获到一个格式错误之前就已经读入了大量字符。例如，对于输入 “(2004, 8, 30)”，只有当读到最后的 “)” 时，我们才能判断遇到了一个格式错误，而一般来说，指望退回这么多字符是不可行的。唯一肯定可以保证的是用 `unget()` 退回一个字符。如果 `operator>>()` 读入一个不合法的 `Date`，如 “(2004,8,32)”，`Date` 的构造函数会抛出一个异常，这会使我们跳出 `operator>>()`。

10.10 一个标准的输入循环

在 10.5 节中，我们学习了如何读写文件。但是，随后就学习了更为深入的错误处理相关内容（10.6 节），因此输入循环还是最初地简单地读取一个文件，从头读到尾的方式。这个假定是合理的，因为我们通常会对每个文件进行独立检查，看其是否有效。但是，我们通常是边读边检查的，下面给出了一个通用的解决策略，假定 `ist` 是一个输入流：

```
for (My_type var; ist>>var; ) { // 一直读到文件结束
    // 或许会检查 var 的有效性
    // 并用 var 来执行什么操作
}
// 我们不太可能从 bad 状态中恢复流，除非必须，否则不用尝试这么做
if (ist.bad()) error("bad input stream");
if (ist.fail()) {
    // 这是一个可接受的终结符吗？
}
// 继续：我们发现到了文件尾
```

也就是说，我们读入一组值，将其保存到变量中，当无法再读入更多值的时候，需要检查流的状态，看是什么原因造成的。类似 10.6 节中的内容，我们可以对这段代码稍加改进，使输入流在发生错误时抛出一个 `failure` 异常，以免我们需要不断检查发生的故障。

```
// 在某处：使 ist 在处于 bad 状态时抛出一个异常
ist.exceptions(ist.exceptions()|ios_base::badbit);
```

我们也可以指定一个字符作为终结符：

```
for (My_type var; ist>>var; ) { // 一直读到文件结束
    // 或许会检查 var 的有效性
    // 并用 var 来执行什么操作
}
if (ist.fail()) { // 使用 ']' 作为终结符与 / 或分隔符
    ist.clear();
    char ch;
    if (!(ist>>ch && ch=='[')) error("bad termination of input");
}
// 继续：我们发现到了文件尾或者找到了一个终结符
```

如果不想要一个特别的终结符，即只接受文件尾作为输入的结束，只需简单地将 `error()` 调用之前的检测语句去掉即可。但是，如果文件包含嵌套结构，那么使用终结符是很有用的，例如，文件由每月的读数组成，每月的读数是由每天读数组成的，而每天的读数是由每小时读数组成的，等等。因此在后面的讨论中都假定使用终结符。

不幸的是，这段代码仍然有些乱。特别是，在读入很多文件的情况下，重复检测终结符是很烦人的。可以定义一个函数来进行处理：

```
// 在某处：使 ist 在处于 bad 状态时抛出一个异常
ist.exceptions(ist.exceptions()|ios_base::badbit);

void end_of_loop(istream& ist, char term, const string& message)
{
    if (ist.fail()) { // 使用 term 作为终结符与 / 或分隔符
        ist.clear();
        char ch;
        if (ist>>ch && ch==term) return; // 所有的都正常
        error(message);
    }
}
```

于是输入循环变为：

```
for (My_type var; ist>>var; ) { // 一直读到文件结束
    // 或许会检查 var 的有效性
    // 用 var 来执行什么操作
}
end_of_loop(ist, ')', "bad termination of file"); // 测试我们是否可以继续
// 继续：我们发现到了文件尾或者找到了一个终结符
```

函数 `end_of_loop()` 什么也不做，除非流处于 `fail()` 状态。我们认为，这样一个输入循环结构足够简单、足够通用，适合很多应用。

10.11 读取结构化的文件

下面我们将应用这个“标准输入循环”来解决一个实际问题。照例，我们还是通过这个例

子来展示有广泛应用范围的设计和编程技术。假定要处理一个温度读数文件，其结构为：

- 文件包含若干年份（包含每月的读数）。
 - 一个年份以“{ year”开始，后跟一个整数，表示年份值，如 1900，然后以“}”结束。
- 一个年份包含若干月份（包含每日的读数）。
 - 一个月份以“{ month”开始，后跟一个三字符形式的月份名，如 jan，然后以“}”结束。
- 一个读数由一个时间和一个温度值组成。
 - 一个读数以“(”开始，后跟日期值、小时值和温度值，最后以“)”结束。

例如：

```
{ year 1990 }
{year 1991 { month jun }}
{ year 1992 { month jan ( 1 0 61.5 ) {month feb (1 1 64) (2 2 65.2) } }
{year 2000
    { month feb (1 1 68) (2 3 66.66) (1 0 67.2)
    {month dec (15 15 -9.2) (15 14 -8.8) (14 0 -2) }
}
```

这种格式有点怪，通常结构化文件的格式都有些特别。现在工业界的发展趋势是，结构化文件变得更有规律、更为层次化（如 HTML 和 XML 文件）。但现实情况是，我们还是极少能控制需要处理的文件的格式。文件的格式就是这个样子，我们要做的就是正确读取其内容。如果格式非常乱，或文件包含太多错误，我们可以编写一个格式转换程序，将文件转换为更适合我们程序的格式。另一方面，我们通常还可以选择数据的内存表示形式以适应程序的需求，因此我们通常可以选择合适的输出格式，来满足特定的需求和偏好。

假定我们只能接受上述给定的温度读数文件格式。幸运的是，其中的年、月等组成部分都是可以自识别的（这有点像 HTML 或 XML 文件）。另一方面，单个读数的格式对合法性检查没有什么帮助。例如，没有什么信息可以帮助我们处理下列情况：用户将日期值和小时值交换；用户使用摄氏度，而程序期望的是华氏度，或者相反。我们必须应对这些情况。

10.11.1 在内存中的表示

应该如何在内存中表示读数呢？一个很直接的做法是定义三个类 Year、Month 和 Reading，与输入准确匹配。显然，Year 和 Month 在处理数据过程中很有用：我们希望比较不同年份的温度，计算每月的平均温度值，比较一年中不同月份的温度，比较不同年份相同月份的温度，将温度读数与日照记录和湿度读数进行匹配，等等。本质上说，Year 和 Month 与我们思考温度和天气的一般方式是吻合的：Month 包含了一个月的信息，而 Year 包含了一年的信息。但是 Reading 呢？它与硬件（如传感器）的底层表示形式吻合。一个 Reading 对象的数据“（日期，小时，温度）”显得很奇怪，而且只在 Month 对象内才有意义。它还是非结构化的：读数不保证按日期或小时顺序给出。基本上，无论何时我们想对读数进行感兴趣的操作时，都要进行排序。

对于温度数据的内存表示，可以作如下假定：

- 如果我们获得了某月的任何一个读数，就很可能会读取该月的其他更多读数。
- 如果我们获得了某日的一个读数，就很可能会读取该日的其他更多读数。

如果情况确实如此，那么就有必要将 Year 表示为 12 个 Month 的 vector，Month 是包含

30 个 Day 的一个向量，而 Day 包含 24 个温度值（每小时一个）。对于很多应用来说，这种方式简单、易于处理。因此，Day、Month 和 Year 都是简单数据结构，只是带有构造函数。既然我们计划在获取温度读数之前就创建 Month 和 Day 来作为 Year 的一部分，那么还需要使用一个“非读数”的概念，来表示某个小时数据还未读入。

```
const int not_a_reading = -7777; // 定义绝对小于 0 的数值
```

类似地，我们引入“非月份”的概念来直接表示未读入数据的月份，以免不得不搜索该月所有日期来确定不包含数据：

```
const int not_a_month = -1;
```

于是三个关键的类可以定义如下：

```
struct Day {
    vector<double> hour {vector<double>(24,not_a_reading)};
};
```

也就是说，Day 包括 24 个小时的读数，每个都被初始化为 not_a_reading。

```
struct Month {
    // 一个月的温度读数
    int month {not_a_month}; // [0: 11], 一月对应 0
    vector<Day> day {32}; // [1: 31], 一个每天温度读数的向量
};
```

为了保持代码简单，这里“浪费”了 day[0]。

```
struct Year {
    // 一年的温度读数，由月份组成
    int year; // 正整数，取值为公元数据
    vector<Month> month {12}; // [0: 11], 一月对应 0
};
```

每个类本质上是一个简单向量，而 Month 和 Year 各有一个成员 month 和 year，分别表明月份和年份。

 这里用到了好几个“常量魔数”（例如 24、32 和 12）。我们试图避免在程序中使用这种文字常量。这里使用的几个常量都是非常基本的（一年有几个月几乎是不会改变的），而且不会在程序其他部分使用。在程序中保留它们主要是为了能够提醒我们注意“常量魔数”的存在所带来的问题。符号常量几乎永远都是更好的选择（参见 7.6.1 节）。使用 32 作为一个月中的天数，肯定要进行合理的解释，此处，32 显然就是“有魔力的”。

为什么不按照下面的方式来写：

```
struct Day {
    vector<double> hour {24,not_a_reading};
};
```

这样的写法可能更简单，但我们可能会得到一个包含两个元素（24 和 -1）的 vector。当我们想要指定 vector 元素个数，而该整数又可以转换为元素类型时，就必须使用() 初始化语法（参见 13.2 节）。

10.11.2 读取结构化的值

可以使用 Reading 类来读取输入，而且更为简单：

```
struct Reading {
    int day;
    int hour;
```

```

    double temperature;
};

istream& operator>>(istream& is, Reading& r)
// 将 is 中的温度数读取到 r 中
// 格式: (3 4 9.7)
// 检查格式, 但不考虑数据有效性
{
    char ch1;
    if (is>>ch1 && ch1!=')' { // 检查是否为 Reading ?
        is.unget();
        is.clear(ios_base::failbit);
        return is;
    }

    char ch2;
    int d;
    int h;
    double t;
    is>>d>>h>>t>>ch2;
    if (!is || ch2!=')') error("bad reading"); // 混乱的读数
    r.day = d;
    r.hour = h;
    r.temperature = t;
    return is;
}

```

基本上, 我们还是先检查格式是否合法, 如果不合法, 我们将文件状态置为 fail(), 并返回。这允许我们尝试通过其他方式读取信息。另一方面, 如果在读取了一些数据后才发现格式错误, 就没有了错误恢复的机会, 我们只能通过 error() 退出。

Month 的输入操作实现大体相同, 只有一点不同: 必须读入任意数目的 Reading 对象, 而不是像 Reading 的 >> 那样只需读取一组固定个数的值:

```

istream& operator>>(istream& is, Month& m)
// 将 is 中的月份数读取到 m 中
// 格式: {month feb...}
{
    char ch = 0;
    if (is>>ch && ch!=')' {
        is.unget();
        is.clear(ios_base::failbit); // 读入 Month 失败
        return is;
    }

    string month_marker;
    string mm;
    is>>month_marker>>mm;
    if (!is || month_marker!="month") error("bad start of month");
    m.month = month_to_int(mm);
    int duplicates = 0;
    int invalids = 0;
    for (Reading r; is>>r; ) {
        if (is_valid(r)) {
            if (m.day[r.day].hour[r.hour] != not_a_reading)
                ++duplicates;
            m.day[r.day].hour[r.hour] = r.temperature;
        }
    }
}

```

```

    else
        ++invalids;
}
if (invalids) error("invalid readings in month",invalids);
if (duplicates) error("duplicate readings in month", duplicates);
end_of_loop(is,'}', "bad end of month");
return is;
}

```

`month_to_int()` 将月份的符号表示（如 `jun`）转换为一个 0 到 11 之间的整型值，这在后面会继续讨论。需要注意的是，代码中使用了 10.10 节中给出的 `end_of_loop()` 来检测终结符。我们对不合法的和重复的 `Readings` 进行计数，计数结果可能对其他人是有用的。

`Month` 的 `>>` 会快速检查 `Reading` 对象的合法性，然后将其存入 `vector`：

```

constexpr int implausible_min = -200;
constexpr int implausible_max = 200;

bool is_valid(const Reading& r)
// 一个粗略的测试
{
    if (r.day<1 || 31<r.day) return false;
    if (r.hour<0 || 23<r.hour) return false;
    if (r.temperature<implausible_min|| implausible_max<r.temperature)
        return false;
    return true;
}

```

最后，我们可以设计 `Year` 的输入操作，与 `Month` 类似：

```

istream& operator>>(istream& is, Year& y)
// 将 is 中的年份数读取到 y 中
// 格式：{year 1972...}
{
    char ch;
    is >> ch;
    if (ch!='{' {
        is.unget();
        is.clear(ios::failbit);
        return is;
    }

    string year_marker;
    int yy;
    is >> year_marker >> yy;
    if (!is || year_marker!="year") error("bad start of year");
    y.year = yy;

    while(true) {
        Month m;           // 每一次循环中都得到一个空的 m 值
        if(!is >> m)) break;
        y.month[m.month] = m;
    }

    end_of_loop(is,'}', "bad end of year");
    return is;
}

```

我们当然想将“烦人的相似”变成就是“相似”，这样就可以将这几段代码合而为一了。但其中有一个非常重要的不同。请看下面的输入循环代码，会出现所期待的结果吗？

```
for (Month m; is >> m; )
    y.month[m.month] = m;
```

也许会得到我们期待的结果，因为这就是目前为止我们设计输入循环的方法，但它是错的。问题在于 `operator>>(istream& is, Month& m)` 并不为 `m` 赋予新值，而只是简单地将 `Reading` 中的数据填入 `m`。因此，反复执行 `is>>m` 会不断将数据填入唯一的一个 `Month` 对象 `m`。这是非常糟糕的！每一个月份实际上都从同年之前月份继承了所有读数，而没有清空该月不包含的读数。因此，每次执行 `is>>m` 时，我们需要一个崭新的、干净的 `Month` 对象。最简单的方法是将 `m` 的定义移入循环内部，这样每次执行 `is>>m` 前都会对其初始化。另一种方式是令 `operator>>(istream& is, Month& m)` 在读入数据之前将 `m` 置空，或者让输入循环完成这一工作。

```
for (Month m; is >> m; {
    y.month[m.month] = m;
    m = Month{}; // 重新初始化 m
}
```

试试下面的程序。

```
// 打开一个输入文件
cout << "Please enter input file name\n";
string iname;
cin >> iname;
ifstream ifs {iname};
if (!ifs) error("can't open input file", iname);

ifs.exceptions(ifs.exceptions() | ios_base::badbit); // 抛出 bad() 状态的异常

// 打开一个输出文件
cout << "Please enter output file name\n";
string oname;
cin >> oname;
ofstream ofs {oname};
if (!ofs) error("can't open output file", oname);
```

```
// 读入任意一个年份的数值
vector<Year> ys;
while(true) {
    Year y; // 每一次循环中获得最新初始化的 Year
    if (!(ifs >> y)) break;
    ys.push_back(y);
}
```

```
cout << "read " << ys.size() << " years of readings\n";
```

```
for (Year& y : ys) print_year(ofs, y);
```

对于 `print_year()` 的实现，可自行练习。

10.11.3 改变表示方法

为了使 `Month` 的 `>>` 能正常工作，我们需要提供一个方法，来读入月份的符号表示。相似地，我们将用符号表示提供匹配的写。一种冗长的表示方法是使用 `if` 语句：

```
if (s == "jan")
    m = 1;
else if (s == "feb")
    m = 2;
...
...
```

这种方法不仅冗长，而且将月份名固化到了程序中。更好的方法是将月份名存入一个表中，使得即便不得不改变符号表示时，也无须改动主程序。我们决定用一个 `vector<string>` 来描述月份的符号表示，另外设计一个初始化函数和一个查找函数。

```
vector<string> month_input_tbl = {
    "jan", "feb", "mar", "apr", "may", "jun", "jul",
    "aug", "sep", "oct", "nov", "dec"
};

int month_to_int(string s)
// s 是一个月份的名字吗？如果是则返回其在 [0:11] 中的索引，否则返回 -1
{
    for (int i=0; i<12; ++i) if (month_input_tbl[i]==s) return i;
    return -1;
}
```

为了避免疑惑，C++ 标准库提供了一种简单的方法来完成相同的工作，参见 16.6.1 节中 `map<string, int>` 的相关内容。

当需要进行输出时，我们将面临一个逆问题。我们在内存中用一个整数表示月份，但输出时希望用符号表示形式。解决方案与输入基本相似，只是把 `string` 到 `int` 的映射表变为 `int` 到 `string` 的映射表：

```
vector<string> month_print_tbl = {
    "January", "February", "March", "April", "May", "June", "July",
    "August", "September", "October", "November", "December"
};

string int_to_month(int i)
// 月份数 [0:11]
{
    if (i<0 || 12<=i) error("bad month index");
    return month_print_tbl[i];
}
```

 好了，你是否真正阅读了全部代码和注释了呢？还是眼睛一眨就跳到末尾了？谨记学习编写高质量代码的最好途径就是阅读大量代码。不管你信不信，本例中我们使用的方法虽很简单，但在没有帮助的情况下领悟其精髓并不容易。读取数据是很基本的，正确编写输入循环（正确初始化用到的变量）是很基本的，转换表示方式也很基本。也就是说，你应该学会这些。唯一的问题是，你是否能学会很好地使用这些技术，以及是否在熬夜之前学会这些基本的技术。

练习

1. 使用 10.4 节中所讨论的方法编写一个程序来处理平面中的点。首先定义包含两个表示坐标的成员 `x` 和 `y` 的数据类型 `Point`。
2. 借助 10.4 节中给出的代码和讨论的技术，提示用户输入 7 个 (x, y) 值对。当用户输入数据时，将其保存在一个名为 `original_points` 的向量中。
3. 打印 `original_points` 中的数据查看结果如何。
4. 打开一个 `ofstream`，将每个点输出到名为 `mydata.txt` 的文件中。在 Windows 平台上，我们建议使用 `.txt` 后缀，这样使用传统的文本编辑器（如写字板）可以很容易地查看文件中的数据。

5. 关闭 `ofstream`, 然后打开一个 `ifstream`, 使其与 `mydata.txt` 关联。从 `mydata.txt` 中读取数据, 保存在一个名为 `processed_points` 的新的向量中。
6. 打印两个向量中的数据元素。
7. 比较两个向量, 如果发现元素数目或值不符, 打印 “`Something's wrong!`”。

思考题

1. 对于大多数现代计算机系统, 处理输入和输出时, 要处理的设备种类有哪些?
2. `istream` 的基本功能是什么?
3. `ostream` 的基本功能是什么?
4. 从本质上讲, 文件是什么?
5. 什么是文件格式?
6. 给出四种需要进行 I/O 的设备类型。
7. 读取一个文件的四个步骤是什么?
8. 写一个文件的四个步骤是什么?
9. 给出四种流状态的名称和定义。
10. 讨论如何解决如下输入问题:
 - a) 用户输入了要求范围之外的值。
 - b) 未读到值 (到达文件末尾)。
 - c) 用户输入了错误类型的数据。
11. 输入通常在哪些方面比输出更难处理?
12. 输出通常在哪些方面比输入更难处理?
13. 我们为什么通常希望将输入输出与计算分离?
14. `istream` 的成员函数 `clear()` 最常用的两个用途是什么?
15. 对于一个用户自定义类型 `X`, `<<` 和 `>>` 通常的函数声明形式如何?

术语

<code>bad()</code>	<code>good()</code>	<code>ostream</code>
<code>buffer</code>	<code>ifstream</code>	output device (输出设备)
<code>clear()</code>	input device (输入设备)	output operator (输出运算符)
<code>close()</code>	<code>input operator</code> (输入运算符)	stream state (流状态)
<code>device driver</code> (设备驱动)	<code>iostream</code>	structured file (结构化文件)
<code>eof()</code>	<code>istream</code>	terminator (终结符)
<code>fail()</code>	<code>ofstream</code>	<code>unget()</code>
<code>file</code> (文件)	<code>open()</code>	

习题

1. 一个文件中保存以空白符间隔的整数, 编写程序求此文件中所有整数之和。
2. 编写程序, 创建一个温度读数文件, 数据格式为 `Reading` 类型, `Reading` 的定义如 10.5 节所述。向文件中填入至少 50 个温度读数。将此程序命名为 `store_temps.cpp`, 读数文件命名为 `raw_temps.txt`。

3. 编写程序，从习题 2 创建的 `raw_temps.txt` 中读取数据，存入一个向量，随后计算数据集中温度的均值和中间值。将此程序命名为 `temp_stats.cpp`。
4. 修改习题 2 中的程序 `store_temps.cpp`，为每个读数附加一个后缀 `c` 或者后缀 `f`，分别表示摄氏度和华氏度。然后修改程序 `temp_stats.cpp`，检测每个温度读数，在存入向量之前将摄氏度转换为华氏度。
5. 编写 10.11.2 节中提到的 `print_year()` 函数。
6. 定义 `Roman_int` 类，保存罗马数字（以 `int` 类型保存），为其定义 `<<` 和 `>>` 运算符。为其定义 `as_int()` 成员函数，返回 `int` 型值，使得对于 `Roman_int` 对象，可以写出语句 `cout << "Roman" << r << "equals" << r.as_int() << '\n';`
7. 修改第 7 章中的计算器程序，使其接受罗马数字而不是阿拉伯数字，例如，`XXI+CIV==CXXV`。
8. 编写程序，接受两个文件名，生成一个新文件，内容为两个输入文件的拼接，即将第二个文件内容拼接到第一个文件内容后面。
9. 两个文件包含已排序的、空白符间隔的单词，编写程序将它们合并，结果文件中单词仍有序排列。
10. 改写第 7 章中的计算器程序，增加“`from x`”命令，使其从文件 `x` 获取输入。增加“`to y`”命令，实现输出到文件 `y`（包括计算结果和错误信息）。设计一系列的测试用例，设计思路如 7.3 节所述，用它们来测试改写后的计算器程序。讨论如何将这两个命令用于计算器程序的测试。
11. 编写程序，对于一个文本文件，找出其中空白符间隔开的所有整数，求它们的和。例如，“`bear: 17 elephants 9 end`” 应该输出 26。

附言

很多计算过程都包含将大量数据从某处移动到另一处的操作，例如，将文件中的文本复制到屏幕，或者将音乐从计算机移动到 MP3 播放器。通常，在迁移过程中还伴随着数据转换。如果数据可以看作值的序列（流），`iostream` 库很适合处理这类任务。在一般的编程工作中，输入输出部分的编码令人吃惊地占据了非常大的比例。这一方面是因为我们（或我们的程序）需要大量数据，另一方面是数据进入程序的入口正是错误多发地带。因此，我们必须努力使 I/O 部分更为简单，并尽量降低有害数据“溜进”程序的几率。

定制输入输出

保持简洁：尽可能地简洁，但不要过度简单化。

——Albert Einstein

在本章中，我们着重介绍如何采用第 10 章所提出的通用 `iostream` 框架来解决特定的输入输出需求和偏好。其中涉及大量较为困难的细节，这一方面是由人类对输入内容的敏感性所决定，另一方面则是由文件使用上的实际限制所决定。本章的最后一个例子会展示一个输入流的设计，它允许指定间隔符集合。

11.1 有规律的与无规律的输入和输出

`iostream` 库，也就是 ISO C++ 标准库的输入输出部分，为文本输入输出提供了一个统一的、可扩展的框架。这里的“文本”指的是任何可以表示为字符序列的数据。这样，当我们讨论输入输出时，可以将 1234 这样的整数也看作文本，因为它可以写成 4 个字符 1、2、3、4。

到目前为止，我们对所有的输入源都是以相同方式处理的。但有时这是不够的，例如，有些文件可能与其他输入源不同（如通信连接），在其中我们可以定位单个字节。类似地，我们还假定输入输出格式完全由对象类型所决定。这是不完全正确的，某些情况下也是不够的。例如，我们常常会指定浮点数输出的数字个数（精度）。本章将介绍一些方法，使我们可以按需求定制输入输出。

作为程序员，我们更喜欢有规律的编程，如一致地处理所有内存对象、以相同方式处理所有输入源、强制使用单一标准对进入和离开系统的对象进行表示，从而写出干净、简单、易于维护而且通常更高效的代码。但是，程序的存在是为了服务于人类，而人类都有自己强烈的偏好。因此，作为程序员，我们必须力争在程序复杂性和满足用户个人偏好间达到平衡。

11.2 格式化输出

人们常常会在意很多输出中的微小细节。例如，对一个物理学家来说，1.25（舍入到小数点后两位数字）与 1.24670477 可能是有很大不同的。而对于一个会计，(1.25) 从法律角度看与 (1.2467) 是不一样的，而与 1.25 则是根本不同的（在金融文件中，括号有时表示亏损，也就是负值）。作为程序员，我们的目标是令输出尽可能地清晰和接近程序“客户”的期望。输出流（`ostream`）提供了很多方法格式化内置类型的输出。对于用户自定义类型，则需要由程序员定义适合的 `<<` 操作。

对于输出，似乎有数不清的细节、优化的余地和不同的选择需要考虑，对于输入，要考虑的类似问题也不少。例如，用来表示小数点的字符（通常是点或逗号）；输出金额数值的方式；输出单词 `true`（或 `vrai` 或 `sandt`）而不是数值 1 来表示真；处理非 ASCII 字符集

(如 Unicode) 的方式；以及限制读入字符串的字符数目等等。除非你需要使用这些功能，否则它们看起来很无趣。因此，我们将这些内容放在手册和专门的著作中，如 Langer 的《Standard C++ IOStreams and Locales》，Stroustrup 的《C++ Programming Language》的第 38 和 39 章，以及《ISO C++ 标准》的第 22 和 27 节。本书只介绍一些最常用的功能和一般性概念。

11.2.1 输出整数

整型值可以输出为八进制（octal，基数为 8 的数制系统）、十进制（decimal，人类常用的数制系统，基数为 10）和十六进制（hexadecimal，基数为 16 的数制系统）。如果你不了解这些数制，请先阅读附录 A.2.1.1，然后再继续学习本章。大多数输出都使用十进制。十六进制多用于输出与硬件相关的信息，原因在于一个十六进制数字精确地表示了 4 位二进制值。因此，两个十六进制数字可以用来表示一个 8 位字节，4 个十六进制数字表示 2 字节的值（常被称为半字），8 位十六进制数则表示 4 字节的值（通常是一个字或一个寄存器的大小）。当 20 世纪 70 年代 C 语言（C++ 的祖先）最初发明之时，表示二进制位更多采用八进制，现在则很少使用了。

我们可以指定（十进制）数 1234 以十进制、十六进制（通常简称为“hex”）或八进制输出：

```
cout << 1234 << "\t(decimal)\n"
    << hex << 1234 << "\t(hexadecimal)\n"
    << oct << 1234 << "\t(octal)\n";
```

字符 “\t” 是制表符“tab”（“tabulation character”的简称），这段代码会输出如下内容：

```
1234 (decimal)
4d2 (hexadecimal)
2322 (octal)
```

符号 `<<hex` 和 `<<oct` 并不输出任何内容，然而 `<<hex` 通知流应该以十六进制输出任何后来的整型值，而 `<<oct` 通知流以八进制输出后来的整数。例如：

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << 1234 << '\n'; // 八进制的基数仍然起作用
```

这段代码会输出：

```
1234 4d2 2322
2322 // 整数将以八进制的形式输出，直到输出格式被改变
```

注意最后一行的输出是一个八进制数。也就是说，`oct`、`hex` 和 `dec`（十进制输出）是持久的（“不变的”）——后来的整数一直按照这种数制输出，直至我们指定新的数制。`hex` 和 `oct` 这种用来改变流的行为的关键字被称为操纵符（manipulator）。

试一试

以十进制、十六进制和八进制输出你的出生年份。为每个值加上标识。利用制表符调整位置使输出按列对齐。然后再输出你的年龄。

一个数值以非十进制显示，总是让人看着有些迷惑。例如，除非我们告诉你，否则你一定会假定 11 表示（十进制）数值 11，而不是 9（八进制数 11 所表示的十进制值），或者 17

(十六进制数 11 所表示的十进制值)。为了解决这个问题，我们可以要求 `ostream` 显示每个整数的基数。例如：

```
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
cout << showbase << dec;      // 显示基数
cout << 1234 << '\t' << hex << 1234 << '\t' << oct << 1234 << '\n';
```

会输出：

```
1234 4d2 2322
1234 0x4d2 02322
```

这样，十进制数将没有前缀，八进制数将带前缀 0，而十六进制数将带前缀 0x (或 0X)。这与 C++ 源程序中的整数文字常量的表示方式是完全一致的。例如：

```
cout << 1234 << '\t' << 0x4d2 << '\t' << 02322 << '\n';
```

如果是十进制输出格式，这段代码会输出：

```
1234 1234 1234
```

你可能已经注意到，与 `oct` 和 `hex` 一样，`showbase` 也是持久的。想去掉其效果的话，可使用 `noshowbase` 操纵符，它会恢复默认效果——输出整数时不显示基数。

对整数输出操纵符总结如下：

整数输出操纵符

<code>oct</code>	使用 8 为基数的（八进制）表示
<code>dec</code>	使用 10 为基数的（十进制）表示
<code>hex</code>	使用 16 为基数的（十六进制）表示
<code>showbase</code>	为八进制加前缀 0，为十六进制加前缀 0x
<code>noshowbase</code>	取消前缀

11.2.2 输入整数

默认情况下，`>>` 假定数值使用十进制表示，但你可以指定读入十六进制或八进制数：

```
int a;
int b;
int c;
int d;
cin >> a >> hex >> b >> oct >> c >> d;
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

如果你键入：

```
1234 4d2 2322 2322
```

上面程序会输出：

```
1234 1234 1234 1234
```

注意，这意味着 `oct`、`dec` 和 `hex` 对输入也是持久的，如同在输出操作中一样。

试一试

完成上面代码片段，形成一个完整程序。先尝试前面给出的输入内容，然后输入下

面的内容：

```
1234 1234 1234 1234
```

解释程序输出的结果。再尝试其他输入，观察输出结果。

你可以让 `>>` 接受前缀 0 和 0x 并正确解释。为了实现这一效果，你需要“复位”所有默认设置，例如：

```
cin.unsetf(ios::dec); // 不再设定十进制显示（这样 0x 可以意味着十六进制）
cin.unsetf(ios::oct); // 不再设定八进制（这样 12 可以意味着 12）
cin.unsetf(ios::hex); // 不再设定十六进制（这样 12 可以意味着 12）
```

流的成员函数 `unsetf()` 将参数中给出的一个或多个标识位复位。现在，对于下面代码

```
cin >>a >> b >> c >> d;
```

如果键入：

```
1234 0x4d2 02322 02322
```

会得到如下输出：

```
1234 1234 1234 1234
```

11.2.3 输出浮点数

如果你直接面对硬件的话，需要使用十六进制（或者八进制）的表示方式。类似地，如果你进行科学计算，就必须处理浮点数的格式。这些类型的处理与整型值的处理方法类似，都是借助于 `iostream` 的操纵符。例如：

```
cout << 1234.56789 << "\t\l\l(defaultfloat)\n"      // \t\l\l 的作用是按列对齐
     << fixed << 1234.56789 << "\t(fixed)\n"
     << scientific << 1234.56789 << "\t(scientific)\n";
```

会输出：

```
1234.57          (general)
1234.567890      (fixed)
1.234568e+003    (scientific)
```

操纵符 `fixed`、`scientific` 和 `defaultfloat` 用来选择浮点数格式。`defaultfloat` 是默认格式（也叫作通用格式（general format））。现在，我们可以这样写：

```
cout << 1234.56789 << '\t'
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
cout << 1234.56789 << '\n';           // 浮点格式是持久的
cout << defaultfloat << 1234.56789 << '\t' // 浮点值输出的默认格式
     << fixed << 1234.56789 << '\t'
     << scientific << 1234.56789 << '\n';
```

会输出：

```
1234.57    1234.567890    1.234568e+003
1.234568e+003                                // scientific 操纵符的效果是持久的
1234.57    1234.567890    1.234568e+003
```

对基本的浮点数格式化输出操纵符总结如下：

浮点数格式

fixed	使用定点表示
scientific	使用尾数和指数表示方式。尾数总在 [1:10) 之间，也就是说，在小数点之前有单个非 0 数字
defaultfloat	在 defaultfloat 的精度范围内自动选择 fixed 或者 scientific 中更为精确的一种表示

11.2.4 精度

默认设置下，defaultfloat 格式用总共 6 位数字来输出一个浮点值。流会选择最适合的格式，~~×~~ 浮点值按 6 位数字（defaultfloat 格式的默认精度）所能表示的最佳近似方式进行舍入。例如：

1234.567 输出为 1234.57

1.2345678 输出为 1.23457

舍入规则采用常用的 4/5 规则：0 到 4 舍，5 至 9 入。注意，浮点格式只对浮点数起作用，于是

1234567 输出为 1234567（因为这是一个整数）

1234567.0 输出为 1.23457e+006

在第二个例子中，ostream 判断出 1234567.0 在 fixed 格式下不能只用 6 位数字输出，因此选择 scientific 格式，保持最为精确的表示形式。基本上，defaultfloat 格式在 scientific 和 fixed 两种格式间进行选择，期望将浮点数以最精确的表示形式呈现给用户，所采用的精度限定为 general 格式的精度——默认为 6 位数字长度。

 试一试

编写代码，三次输出浮点数 1234567.89，分别采用 defaultfloat、fixed 和 scientific 格式。哪种格式呈现给用户最精确的表示形式？解释为什么。

程序员可以使用操纵符 setprecision() 来设置精度，例如：

```
cout << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
cout << defaultfloat << setprecision(5)
    << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
cout << defaultfloat << setprecision(8)
    << 1234.56789 << '\t'
    << fixed << 1234.56789 << '\t'
    << scientific << 1234.56789 << '\n';
```

会输出（注意舍入）：

```
1234.57    1234.567890    1.234568e+003
1234.6 1234.56789  1.23457e+003
1234.5679 1234.56789000 1.23456789e+003
```

几种格式的精度分别定义为：

浮点数精度

defaultfloat	精度就是数字的个数
scientific	精度为小数点之后数字的个数
fixed	精度为小数点之后数字的个数

一般使用默认的精度格式（精度为 6 的 `defaultfloat` 格式），除非有特殊原因——一个常见的原因是“因为我们需要更为精确的输出”。

11.2.5 域

使用 `scientific` 和 `fixed` 格式，程序员可以精确控制一个值输出所占用的宽度。显然，这对于打印表格这类应用来说很有用。整数输出也有类似的机制，称为域（field）。你可以使用“设置域宽度”操纵符 `setw()` 精确指定一个整数或一个字符串输出占用多少个位置。例如：

```
cout << 123456           // 不使用域
<< '|' << setw(4) << 123456 << '|'
<< setw(8) << 123456 << '|'
<< 123456 << "\n";      // 域的大小不会持久化
```

会输出：

```
123456|123456| 123456|123456|
```

首先注意第三个 123456 之前的两个空格，这就是我们所期望的效果——一个 6 位数字的数占用一个 8 个字符的域。但是，当你指定一个 4 个字符的域时，123456 不会被截取来适应域宽。为什么不截取呢？`|1234|` 或 `|3456|` 对于宽度为 4 字符的域来说都是适合的，但它们完全改变了要输出的值，而且没有给用户任何警告信息。`ostream` 是不会这样做的，相反，它会打破输出格式。坏的格式总比“坏的输出数据”更好些。而且在使用域最多的应用中（例如打印表格），“溢出”问题是很容易注意到的，因此能被修正。

域也可作用于浮点数和字符串，例如：

```
cout << 12345 << '|' << setw(4) << 12345 << '|'
<< setw(8) << 12345 << '|' << 12345 << "\n";
cout << 1234.5 << '|' << setw(4) << 1234.5 << '|'
<< setw(8) << 1234.5 << '|' << 1234.5 << "\n";
cout << "asdfg" << '|' << setw(4) << "asdfg" << '|'
<< setw(8) << "asdfg" << '|' << "asdfg" << "\n";
```

会输出：

```
12345|12345| 12345|12345|
1234.5|1234.5| 1234.5|1234.5|
asdfg|asdfg| asdfg|asdfg|
```

注意，域的宽度不是持久的。在上述 3 个例子中，第一个和最后一个例子值的输出方式是默认的“它需要多少位置就占用多少位置”格式。换句话说，除非你在语句中直接在输出操作之前设置域宽，否则不会有域的限制。

试一试

编写程序，创建一个简单的表格，包括你自己和至少 5 位朋友的姓、名、电话号码、email 地址等信息。试验不同的域宽，直至表格输出形式达到你满意的程度为止。

11.3 打开和定位文件

 从 C++ 程序的角度看，文件是操作系统提供的一个抽象。如 10.3 节所述，一个文件就是一个从 0 开始编号的简单的字节序列：



问题是我们如何访问这些字节。如果使用 `iostream`, 访问方式很大程度上在我们打开文件将其与一个流相关联时就确定了。流的属性决定了文件打开后我们可以对它执行哪些操作, 以及这些操作的意义。最简单的一个例子是, 如果打开文件关联至一个 `istream`, 我们可以从文件读取数据, 而使用 `ostream` 打开文件的话, 我们可以向文件写入数据。

11.3.1 文件打开模式

可以使用多种模式打开文件。默认情况下, 用 `ifstream` 打开的文件用于读, 用 `ofstream` 打开的文件用于写, 这满足了大多数一般需求。但是, 你还可以选择其他方式:

文件流打开模式

<code>ios_base::app</code>	追加模式 (即添加在文件末尾)
<code>ios_base::ate</code>	“末端”模式 (打开文件并定位到文件尾)
<code>ios_base::binary</code>	二进制模式——注意系统特有的行为
<code>ios_base::in</code>	读模式
<code>ios_base::out</code>	写模式
<code>ios_base::trunk</code>	将文件截为长度 0

可以在文件名之后指定文件模式, 例如:

```
ofstream of1 {name1};           // 默认设置为 ios_base::out
ifstream if1 {name2};           // 默认设置为 ios_base::in

ofstream ofs {name, ios_base::app}; // 带 io_base::out 模式的默认设置的输出流
fstream fs {"myfile", ios_base::in|ios_base::out}; // 同时带 in 和 out 模式的流
```

后一个例子中的 “|” 是 “位或” 运算符 (参见附录 A.5.5), 可用于组合多个模式。`app` 模式常用于写日志文件, 因为你总是将新的日志追加到文件末尾。

在每个例子中, 打开文件的确切效果依赖于操作系统, 而且如果操作系统不能使用某种特定的模式打开文件的话, 流可能会进入非 `good()` 状态。

```
if (!fs) // 糟糕: 我们不能用这种模式打开文件
```

以读模式打开一个文件, 最常见的失败原因是文件不存在 (至少文件名不是我们所指定的那样):

```
ifstream ifs {"readings"};
if (!ifs) // 错误: 不能打开文件 "readings" 读取数据
```

在本例中, 我们猜测是拼写错误导致了文件打开失败。

注意, 如果以写模式打开一个文件, 而文件不存在的话, 通常操作系统会创建一个新文件, 但如果以读模式打开一个不存在的文件, 就不会创建新文件 (这实际上是很幸运的)。

```
ofstream ofs {"no-such-file"}; // 创建名为 "no-such-file" 的新文件
ifstream ifs {"no-file-of-this-name"}; // 错误: ifs 将处于非 good() 状态
```

不要对文件打开模式自作聪明。操作系统对 “异常模式” 的处理无法保证一致性。只要情况允许, 就应坚持只读取由 `istream` 打开的文件, 只写入到由 `ostream` 打开的文件中。

11.3.2 二进制文件



在内存中，我们可以将整数 123 表示为一个整型值或者一个字符串值，如下所示：

```
int n = 123;
string s = "123";
```

在第一条语句中，123 保存为一个（二进制）数，与所有其他整型值占用相同大小的内存空间（在 PC 上是 4 字节，32 位）。即便我们处理数值 12345，仍然占用 4 个字节。在第二条语句中，123 保存为一个 3 个字符的字符串。如果我们处理的是 12345，则保存为字符串 "12345" 需占用 5 个字符（还需要加上管理字符串所需的固定开销）。这种差异如下图所示（可以看到，使用普通的十进制和字符表示方式，不如使用在计算机内部采用的二进制表示方式）：

123 保存为字符:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>?</td><td>?</td><td>?</td><td>?</td><td>?</td></tr></table>	1	2	3	?	?	?	?	?
1	2	3	?	?	?	?	?		
12345 保存为字符:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>?</td><td>?</td><td>?</td></tr></table>	1	2	3	4	5	?	?	?
1	2	3	4	5	?	?	?		
123 保存为二进制:	<table border="1"><tr><td>123</td><td></td></tr></table>	123							
123									
12345 保存为二进制:	<table border="1"><tr><td>12345</td><td></td></tr></table>	12345							
12345									

当我们使用字符表示方式时，必须使用特定字符表示数值的结束，就像我们在纸上书写数值一样：123456 是一个数，而 123 456 是两个数。在纸上书写，我们使用空格来表示数值的结束。在计算机内存中，我们也可以这么做：

123456 保存为字符:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>?</td></tr></table>	1	2	3	4	5	6	?
1	2	3	4	5	6	?		
123 456 保存为字符:	<table border="1"><tr><td>1</td><td>2</td><td>3</td><td></td><td>4</td><td>5</td><td>6</td></tr></table>	1	2	3		4	5	6
1	2	3		4	5	6		

固定长度的二进制表示方式（比如 int 型值）和变长的字符串表示方式（比如 string 类型）之间的差别在文件中也有体现。默认情况下，iostream 使用字符表示方式。也就是说，istream 从文件读取字符序列，并将其转换为所需类型的对象。而 ostream 将指定类型的对象转换为字符序列，然后写入文件。但是，我们可以令 istream 和 ostream 将对象在内存中对应的字节序列简单地复制到文件。这称为二进制 I/O，通过在打开文件时指定 ios_base::binary 模式来实现。下面的例子展示了如何读写二进制整数文件，涉及“二进制”处理的代码后面给出详细解释：

```
int main()
{
    // 以二进制文件读取模式打开一个 istream
    cout << "Please enter input file name\n";
    string iname;
    cin >> iname;
    ifstream ifs {iname,ios_base::binary};           // 注意：流模式
    // binary 告知流不要自作聪明地处理这些字节
    if (!ifs) error("can't open input file ",iname);

    // 以二进制文件写入模式打开一个 ostream
    cout << "Please enter output file name\n";
    string oname;
    cin >> oname;
    ofstream ofs {oname,ios_base::binary};           // 注意：流模式
```

```

// binary 告知流不要自作聪明地处理这些字节
if (!ofs) error("can't open output file ",oname);

vector<int> v;

// 从二进制文件中读取
for(int x; ifs.read(as_bytes(x),sizeof(int)); ) // 注意：读入字节
    v.push_back(x);

// 对 v 进行处理

// 写入到二进制文件
for(int x : v)
    ofs.write(as_bytes(x),sizeof(int)); // 注意：写入字节
return 0;
}

```

打开二进制文件是通过指定 `ios_base::binary` 流模式实现的。

```

ifstream ifs {iname, ios_base::binary};

ofstream ofs {oname, ios_base::binary};

```

在上述例子中，我们使用相对复杂但也更为紧凑的二进制表示方式。当我们从面向字符的 I/O 转向二进制 I/O 时，要放弃常用的 `>>` 和 `<<` 操作符。这两个操作符按默认约定将值转换为字符序列（如，字符串 "asdf" 转换为字符 a、s、d、f，整数 123 转换为字符 1、2、3）。如果我们需要的就是这些，那么也就不必使用二进制了，因为默认模式已经够用了。只有在默认模式不能满足需求时，我们才需要使用二进制文件。我们使用二进制模式，就是告知流不要自作聪明地处理字节序列。

我们如何处理 `int` 型值才是“聪明的”？显然是用 4 个字节存储 4 字节宽的 `int` 型值，也就是说，我们可以查看 `int` 型值在内存中的表示方式（4 个字节的序列），并直接将这些字节传输到文件。随后，我们就可以用同样的方式读回这些字节重组出 `int` 值：

```

ifs.read(as_bytes(i),sizeof(int)) // 注意：读字节
ofs.write(as_bytes(v[i]),sizeof(int)) // 注意：写字节

```

`ostream` 的 `write()` 函数和 `istream` 的 `read()` 函数都接受两个参数：地址（这里用函数 `as_bytes()` 获取）和字节（字符）数量（这里我们用运算符 `sizeof` 获得）。对于我们读 / 写的值，地址参数指向保存它的内存区域的第一个字节。例如，如果我们处理一个 `int` 型值 1234，其内存区域中按以下方式保存这样的 4 个字节（十六进制表示）：00、00、04、d2



函数 `as_bytes()` 可以用来获取对象存储区域的第一个字节。它可以定义如下（其中用到了我们还未介绍的语言特性，参见 12.8 节和 14.3 节）：

```

template<class T>
char* as_bytes(T& i) // 将 T 视为一个字节序列
{
    void* addr = &i; // 得到保存对象的内存区域的第一个字节的地址
    return static_cast<char*>(addr); // 将内存视为多个字节
}

```

使用 `static_cast` 的（不安全）类型转换是必需的，我们需要用它来获得一个变量的“原始字节表示”。地址的概念我们会在第 12 章和第 13 章进行详细介绍。在这里，我们只展示如何将内存中的对象按字节序列的方式处理，供 `read()` 和 `write()` 使用。

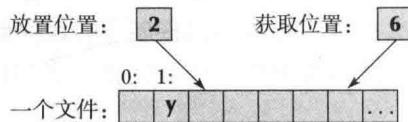
这种二进制 I/O 方式有些困难和复杂，而且容易出错。但是作为程序员，我们不是总有选择文件格式的自由。因此，偶尔我们必须使用二进制文件，只是因为我们要读写的文件的制作者选择了二进制格式。另外，也有可能使用非字符表示方式是一种更好的选择。典型的例子是图像和声音文件，它们都没有适合的字符表示方式：一幅照片或者一段音乐本质上就是一个比特包。

`iostream` 库默认的字符 I/O 是可移植的、人类可读的，而且很好地被类型系统所支持。如果条件允许，请尽量使用字符 I/O，除非不得已，否则不要使用二进制 I/O。

11.3.3 在文件中定位

只要有可能，请尽量使用从头至尾的文件读写方式。这是最简单，也最不容易出错的方式。很多时候，当你觉得必须对文件进行修改时，最好的方法是创建一个新的文件。

但是，如果你必须使用文件定位功能的话，C++ 也支持在文件中定位到指定位置以进行读写。基本上，每个以读方式打开的文件，都有一个“读 / 获取位置”，而每个以写方式打开的文件，都有一个“写 / 放置位置”。



使用方法如下：

```
fstream fs {name};      // 打开文件进行输入输出
if (!fs) error("can't open ",name);

fs.seekg(5);           // 移动读位置 (g 表示“获取”) 到 5 (第 6 个字符处)
char ch;
fs>>ch;              // 进行读操作，并增加读位置
cout << "character[5] is " << ch << '(' << int(ch) << ")\n";

fs.seekp(1);           // 移动写位置 (p 表示“放置”) 到 1
fs<<'y';             // 进行写操作，并增加写位置
```

注意，`seekg()` 和 `seekp()` 增加了它们的相对位置，所以图中表示的是程序执行后的状态。

请小心：这段代码中在文件定位之前进行了运行时错误检测，这是必要的。另外特别要注意的是，如果你试图定位（用 `seekg()` 或者 `seekp()`）到文件尾之后，结果如何是未定义的，不同操作系统会表现出不同行为。

11.4 字符串流

你可以将一个 `string` 对象作为 `istream` 的源，或者 `ostream` 的目标。从一个字符串读取内容的 `istream` 对象称为 `istringstream`，保存字符并将其写入字符串的 `ostream` 对象称为 `ostringstream`。例如，从字符串提取数值时，`istringstream` 就很有用：

```
double str_to_double(string s)
    // 如果可能，将字符转换为浮点数
{
    istringstream is {s};           // 定义一个流来从 s 中读出
    double d;
    is >> d;
    if (!is) error("double format error: ", s);
    return d;
}

double d1 = str_to_double("12.4");           // 测试
double d2 = str_to_double("1.34e-3");
double d3 = str_to_double("twelve point three"); // 会调用 error()
```

如果你试图从一个 `istringstream` 流的字符串尾之后读取字符，`istringstream` 流会进入 `eof()` 状态。这意味着你可以将“标准输入循环”应用于 `istringstream` 流，实际上一个 `istringstream` 流就是一个真正的 `istream`。

相反，对于要求一个简单字符串参数的系统，如 GUI 系统（参见 21.5 节），`ostringstream` 可用于格式化输出来生成单一字符串。例如：

```
void my_code(string label, Temperature temp)
{
    ...
    ostringstream os;           // 生成一条消息的流
    os << setw(8) << label << ":" 
        << fixed << setprecision(5) << temp.temp << temp.unit;
    someobject.display(Point(100,100), os.str().c_str());
    ...
}
```

`ostringstream` 的成员函数 `str()` 返回由输出操作到 `ostringstream` 对象的内容构成的字符串。`c_str()` 是 `string` 的成员函数，它返回很多系统接口所要求的 C 风格字符串。

`stringstream` 通常用于将真实 I/O 和数据处理分离。例如，`str_to_double()` 的 `string` 参数通常来自一个文件（比如一个 Web 日志）或键盘。类似地，我们在 `my_code()` 中生成的消息最终会输出到屏幕的某个区域。再如，在 11.7 节中，我们使用 `stringstream` 来过滤输入中不希望出现的字符。因此，`stringstream` 可以看作一种剪裁 I/O 以适应特殊需求和偏好的机制。

`ostringstream` 的一个简单应用是连接字符串，例如：

```
int seq_no = get_next_number();           // 获取日志文件的编号
ostringstream name;
name << "myfile" << seq_no << ".log"; // 例如，myfile17.log
ofstream logfile{name.str()};            // 例如，打开 myfile17.log
```

通常情况下，我们用一个 `string` 来初始化 `istringstream`，然后用输入操作从该字符串中读取字符。相反，我们通常用一个空字符串初始化 `ostringstream`，然后用输出操作向其中填入字符。有一种更为直接的方法来访问 `stringstream` 中的字符：`ss.str()` 返回 `ss` 的字符串的一个拷贝，而 `ss.str(s)` 则将 `ss` 的字符串设置为 `s` 的一个拷贝。这种方法在某些情况下很有用，11.7 节展示了一个使用 `ss.str(s)` 的例子，体现了它的用处。

11.5 面向行的输入

`>>` 操作符按照对象类型的标准格式读取输入，保存到对象中。例如，当读取一个 `int` 型

对象时，`>>`会一直读取数字，直至遇到一个非数字的字符为止；当读取一个 `string` 时，`>>`会一直读取字符，直至遇到空白符为止。标准库 `istream` 库也提供了读取单个字符和整行内容的功能。考虑下面代码：

```
string name;
cin >> name;           // 输入: Dennis Ritchie
cout << name << '\n'; // 输出: Dennis
```

如果希望一次读取整行内容，随后再决定如何从中格式化输入数据，那么我们应该怎么做呢？可以使用函数 `getline()`，例如：

```
string name;
getline(cin, name);      // 输入: Dennis Ritchie
cout << name << '\n'; // 输出: Dennis Ritchie
```

现在我们已经获得整行内容了。我们为什么要这么做呢？一个很好的答案是：“因为我们需要做一些 `>>` 做不了的事。”通常，一个不好的答案是：“因为用户输入的就是一整行。”如果这是你能想到的最佳答案的话，还是继续使用 `>>` 吧，因为一旦读入了一整行，通常情况下你就必须自己来分析输入内容，例如：

```
string first_name;
string second_name;
stringstream ss {name};
ss>>first_name;          // 输入 Dennis
ss>>second_name;         // 输入 Ritchie
```

显然，直接用 `>>` 将读入字符串，存入 `first_name` 和 `second_name` 更简单些。

使用整行输入的一个常见原因是，默认的空白符不符合我们的要求。有时，我们可能需要将换行符同其他空白符区别对待。例如，与一个电脑游戏的文本交互中，可能将一行作为一句话，而不使用习惯的标点符号。

```
go left until you see a picture on the wall to your right
remove the picture and open the door behind it. take the bag from there
```

对于此例，我们可以先读入一行，然后从中提取单个单词。

```
string command;
getline(cin, command); // 读入一行

stringstream ss {command};
vector<string> words;
for (string s; ss>>s; )
    words.push_back(s); // 提取单个单词
```

但是，只要我们有其他解决方法，还是尽量使用习惯的标点符号而不是换行。

11.6 字符分类

 通常，我们按习惯格式读入整数、浮点数、单词等。但是，我们可以（有时是必须）在更低的抽象层上读入单个字符。这样做在编程上需要做更多工作，但我们可以对输入有完全的控制。回顾一下表达式单词分析问题（7.8.2 节），假如我们希望将 $1+4*x \leq y/z^5$ 分解为 11 个单词：

$1 + 4 * x \leq y / z ^ 5$

我们可以用 `>>` 读入数值，但试图以字符串类型读入标识符时，就会导致 `x<=y` 被作为一个

字符串读入（因为 < 和 = 不是空白符），z* 也是如此（因为 * 也不是一个空白符）。我们可以写出如下代码实现正确的单词分解：

```
for (char ch; cin.get(ch); ) {
    if (isspace(ch)) { // 如果 ch 是一个空白符
        // 什么也不做（也就是说，跳过空白符）
    }
    if (isdigit(ch)) {
        // 读入一个数值
    }
    else if (isalpha(ch)) {
        // 读入一个标识符
    }
    else {
        // 处理操作符号
    }
}
```

函数 `istream::get()` 读入单个字符，赋予它的参数。它不跳过空白符。与 `>>` 类似，`get()` 返回其 `istream` 对象的引用，便于我们检测其状态。

当我们采用逐个字符读取方式时，通常需要对字符进行分类：这个字符是数字吗？这个字符是大写字母吗？等等。下面是实现字符分类的标准库函数：

字符分类

<code>isspace(c)</code>	c 是空白符吗 (' '、'\t'、'\n'，等等) ?
<code>isalpha(c)</code>	c 是字母吗 ('a' .. 'z'、'A' .. 'Z') (注意：不包括 '_') ?
<code>isdigit(c)</code>	c 是十进制数字吗 ('0' .. '9') ?
<code>isxdigit(c)</code>	c 是十六进制数字吗 (十进制数字或者 'a' .. 'f'、'A' .. 'F') ?
<code>isupper(c)</code>	c 是大写字母吗？
<code>islower(c)</code>	c 是小写字母吗？
<code>isalnum(c)</code>	c 是字母或十进制数字吗？
<code>iscntrl(c)</code>	c 是控制字符吗 (ASCII 码 0..31 和 127) ?
<code>ispunct(c)</code>	c 是标点 (除字母、数字、空白符或不可见控制字符之外的字符) 吗？
<code>isprint(c)</code>	c 是可打印字符吗 (ASCII 字符 '!' .. '~') ?
<code>isgraph(c)</code>	c 是字母、十进制数字或者标点吗 (注意：不包括空白符) ?

注意，多个字符分类可以用“或”运算符 (`||`) 进行组合。例如，`isalnum(c)` 意味着 `isalpha(c)||isdigit(c)`，也就是说，“c 是一个字母或者一个数字吗？”

另外，标准库还提供了另外两个有用的函数，用来转换大小写：

字符大小写

<code>toupper(c)</code>	c 或者 c 对应的大写字母
<code>tolower(c)</code>	c 或者 c 对应的小写字母

如果你想忽略大小写的话，这两个函数很有用。例如，用户输入 `Right`、`right` 和 `rigHT` 很可能是想表示相同的意思（`rigHT` 很可能是不小心误按了 Caps Lock 键）。对这些字符串的每个字符都应用 `tolower()` 后，所有字符串都变为了 `right`。我们可以为任何字符串定义 `tolower` 函数：

```
void tolower(string& s)      // 将 s 置为小写格式
{
    for (char& x : s) x = tolower(x);
}
```

参数传递上我们采用了传引用方式（参见 8.5.5 节），以便函数能真正改变实参字符串。如果我们希望保持原字符串的内容不变，可以编写一个函数，创建原字符串的一个小写拷贝。在处理这类问题时，使用 `tolower()` 比使用 `toupper()` 更好些。因为对于某些自然语言如德语，并不是所有小写字母都有对应的大写字母，因此前者能获得更好的效果。

11.7 使用非标准分隔符

本节提供一个接近实际的例子，它使用 `iostream` 库解决一个真实问题。当我们读入字符串时，是以空白符作为默认分隔符的。不幸的是，`istream` 没有提供自定义分隔符的功能，也不能直接改变 `>>` 读入字符串的方式。于是，如果我们需要定义其他空白符，应该怎么做呢？回顾 4.6.3 节中的例子，我们读入“单词”并进行比较。那些单词都是以空白符分隔的，因此，如果我们输入

As planned, the guests arrived; then,

我们会得到这些“单词”：

```
As
planned,
the
guests
arrived;
then,
```

我们在字典中是找不到“`planned,`”和“`arrived;`”这些字符串的，它们并不是单词。它们实际上是由单词加上毫无关系的、分散注意力的标点字符构成的。而在大多数场合下，我们是应该将标点与空白符等同对待的。那么该如何去掉这些标点呢？我们可以逐个处理字符，将标点字符删除或者转换为空白符，随后再从“清理干净的”输入中读取数据：

```
string line;
getline(cin,line);      // 整行读入
for (char& ch : line)  // 将每个标点字符替换为一个空格
    switch(ch) {
        case ',': case '.': case '!': case '?': case ';':
            ch = ' ';
    }

stringstream ss(line);      // 使用 istream ss 读取整行
vector<string> vs;
for (string word; ss>>word; ) // 读取不带标点字符的单词
    vs.push_back(word);
```

同样是前面给出的输入，以下代码会得到我们想要的单词：

```
As
planned
the
guests
arrived
then
```

不幸的是，这段代码有些乱，而且是专用而非通用的。如果对另外一个问题，标点集发生变化，我们又该怎么办呢？下面我们提出一种更为通用、更为有效的从输入流中删除不需要字符的方法。这种方法应该是怎样的呢？我们希望使用这一功能的用户程序是什么样的呢？考虑下面的代码：

```
ps.whitespace(";,:."); // 将分号、冒号、逗号和点都作为空白符处理
for(string word; ps>>word; )
    vs.push_back(word);
```

我们如何才能定义一个像 `ps` 这样的流呢？基本思想是先从一个普通输入流读入单词，然后使用用户指定的“空白符”来处理输入内容，也就是说，我们并不将“空白符”交给用户，我们只是用它们来分隔单词。例如：

`as.not`

应该是两个单词

`as
not`

我们可以定义一个类来实现上述功能。这个类必须从一个 `istream` 读取数据，而且要有一个 `>>` 操作符，能像 `istream` 一样工作，唯一的差别是我们可以告知它哪些字符作为空白符。简单起见，我们不支持默认空白符（空格、换行等）。也就是说，我们只允许用户指定非标准的空白符。我们也不提供从输入流中完全删除指定字符的功能，只是将它们转换为标准空白符。我们将这个类命名为 `Punct_stream`：

```
class Punct_stream { // 和 istream 一样，但是用户可以增加空白符集合
public:
    Punct_stream(istream& is)
        : source{is}, sensitive{true} {}

    void whitespace(const string& s) // 定义 s 为空白符集
        { white = s; }
    void add_white(char c) { white += c; } // 加入到空白符集
    bool is_whitespace(char c); // c 在空白符集中？
    void case_sensitive(bool b) { sensitive = b; }
    bool is_case_sensitive() { return sensitive; }

    Punct_stream& operator>>(string& s);
    operator bool();

private:
    istream& source; // 符号源
    istringstream buffer; // 使用 buffer 处理格式
    string white; // 被视为“空白符”的符号
    bool sensitive; // 该 stream 是否大小写敏感?
};
```

如上面示例代码所示，基本思想是从 `istream` 中一次读取一行，将指定的空白符转换为空格，然后使用 `istringstream` 完成格式化。除了处理用户自定义空白符外，我们还为 `Punct_stream` 实现了一个相关的功能：我们可以通过 `case_sensitive()` 要求它将大小写敏感的输入转换为大小写不敏感的输入。例如，我们可以要求 `Punct_stream` 将

`Man bites dog!`

转换为

```
man
bites
dog
```

`Punct_stream` 的构造函数接受一个 `istream` 参数作为字符输入源，并将其命名为 `source`。构造函数还将流默认设置为大小写敏感的。下面代码可以令 `Punct_stream` 从 `cin` 读入字符，将分号、冒号和点作为空白符，并将所有字符转换为小写：

```
Punct_stream ps {cin};      // ps 从 cin 中读入
ps.whitespace(";::.");    // 分号、冒号和点都是空白符
ps.case_sensitive(false);  // 大小写不敏感
```

显然，最有趣的操作是输入操作符 `>>`，这也是目前为止最难以实现的。基本策略是从 `istream` 读取一整行，存入一个名为 `line` 的字符串，然后将所有自定义空白符转换为空格符 (' ')。完成后，我们将 `line` 放入名为 `buffer` 的 `istringstream` 中。现在就可以使用识别一般空白符的 `>>` 从 `buffer` 中读取数据了。实际代码比上述过程稍微复杂一些，因为从 `buffer` 中读取数据直接就可以进行，但只有在它为空的情况下，才能向其写入内容。

```
Punct_stream& Punct_stream::operator>>(string& s)
{
    while (!(buffer>>s)) {           // 尝试从 buffer 中读取
        if (buffer.bad() || !source.good()) return *this;
        buffer.clear();

        string line;
        getline(source,line);          // 从 source 中读入一行

        // 按需进行字符替换
        for (char& ch : line)
            if (is whitespace(ch))
                ch = ' ';               // 替换为空格
            else if (!sensitive)
                ch = tolower(ch);        // 替换为小写符号

        buffer.str(line);             // 将字符串放入到流中
    }
    return *this;
}
```

我们逐行分析一下这段程序。先看一下这句有点不寻常的代码：

```
while (!(buffer>>s)) {
```

如果名为 `buffer` 的 `istringstream` 中存有字符，读操作 `buffer>>s` 就可以进行，`s` 会收到“空白符”分隔的单词，随后就没有什么可做的了。只要 `buffer` 中有我们可以读取的字符，这个过程就会发生。但是，当 `buffer>>s` 失败时，也就是 `!(buffer>>s)` 为真时，我们必须利用 `source` 中内容将 `buffer` 重新填满。注意读操作 `buffer>>s` 是在一个循环中，当我们尝试重新填充 `buffer` 后，会再次尝试这个读操作，因此有如下代码：

```
while (!(buffer>>s)) {           // 尝试从 buffer 中读取
    if (buffer.bad() || !source.good()) return *this;
    buffer.clear();

    // 重新填充 buffer
}
```

如果 `buffer` 处于 `bad()` 状态，或者 `source` 有问题的话，我们将放弃读取操作。否则，

我们清空 buffer，再次尝试。我们清空 buffer 的原因是，只有在读失败的情况下，通常是 buffer 遇到 eof() 时，我们才进入“重新填充循环”，而这意味着 buffer 中没有供我们读取的字符。处理流状态总是很麻烦的，而且常常是微妙错误的根源，需要冗长的调试过程来消除。幸运的是，重填循环的剩余部分很简单：

```
string line;
getline(source, line);           // 从 source 中读入一行

// 按需进行字符替换
for (char& ch : line)
    if (is whitespace(ch))
        ch = ' ';                // 替换为空格
    else if (!sensitive)
        ch = tolower(ch);         // 替换为小写符号

buffer.str(line);               // 将字符串放入到流中
```

我们先读入一整行到 line，然后检查其中每个字符，看是否需要改变。is whitespace() 是 Punct_stream 的成员函数，我们随后再定义它。tolower() 是标准库函数，其功能显然是将字母转换为小写，如将 A 转换为 a（参见 11.6 节）。

一旦我们正确处理完 line 中内容，就需要将其存入 istream。buffer.str(line) 完成这一工作，这条语句可以读作“将 istream 对象 buffer 的字符串设置为 line 的内容”。

注意，使用 getline() 从 source 读入一行字符后，我们“忘记了”检测它的状态。实际上是不必那么做，原因是最终会到达位于循环顶部的 !source.good() 语句，这时就会进行检测了。

这里我们仍然将流自身的引用——*this 作为 >> 的返回结果，参见 12.10 节。

测试空白符的部分很容易实现，我们只需将输入字符与空白符集中所有指定空白符一一进行比较即可：

```
bool Punct_stream::is whitespace(char c)
{
    for (char w : white)
        if (c==w) return true;
    return false;
}
```

记住，我们将处理默认空白符（如换行和空格）的工作留给 istream 来做了，因此无须对这类空白符做特殊处理。

下面，只剩下最后一个神秘的函数了：

```
Punct_stream::operator bool()
{
    return !(source.fail() || source.bad() && source.good());
}
```

istream 的一种习惯用法是测试 >> 的结果，例如：

```
while (ps>>s) { /* . . . */ }
```

这意味着我们需要一种方法将 ps>>s 的结果当作布尔值来进行检查。但 ps>>s 结果是一个 Punct_stream，因此我们需要一种方法将一个 Punct_stream 隐式转换为一个 bool 值。这就是 Punct_stream 的运算符 bool() 所做的事情。Punct_stream 的名为 opeartor bool() 的成员函数

定义了流到 `bool` 类型的转换。特别地，它在 `Punct_stream` 上的操作成功时返回 `true`。

现在我们就可以写程序了。

```
int main()
{
    // 给定文本输入，产生一个该文本中所有单词的升序列表
    // 忽略标点符号和大小写的区别
    // 去掉输出中的重复结果
{
    Punct_stream ps {cin};
    ps.whitespace(";,.?!()\"{}</&$@#%^*|~"); // 注意，\“表示在字符串中的”
    ps.case_sensitive(false);

    cout << "please enter words\n";
    vector<string> vs;
    for (string word; ps>>word; )
        vs.push_back(word); // 读入单词

    sort(vs.begin(),vs.end()); // 按字典序进行排序
    for (int i=0; i<vs.size(); ++i) // 写入字典
        if (i==0 || vs[i]!=vs[i-1]) cout << vs[i] << '\n';
}
}
```

这个程序会将输入中出现的单词排序输出。测试语句

```
if (i==0 || vs[i]!=vs[i-1])
```

会去掉重复单词。给这个程序下面的输入内容：

```
There are only two kinds of languages: languages that people complain
about, and languages that people don't use.
```

程序会输出

```
about
and
are
complain
don't
kind
languages
of
only
people
that
there
two
use
```

为什么会得到 `don't` 而不是 `dont` 呢？因为我们并没有将单引号指定为空白符。

 注意：`Punct_stream` 在很多重要和有用方面都与 `istream` 相似，但它的确不是一种 `istream`。例如，我们不能使用 `rdstate()` 来获取流状态，`eof()` 也没有定义，而且我们也不必为针对整数的 `>>` 而烦恼（`Punct_stream` 只是用来处理字符串的）。重要的是，对于一个期望 `istream` 参数的函数，我们不能将一个 `Punct_stream` 传递给它。我们可以使 `Punct_stream` 真的成为 `istream` 吗？当然可以，但现在我们所拥有的编程经验、设计思想和语言工具还不足以实现这一目标（如果你以后希望回过头再来完成这个目标——当然是很久以后的事情，你还必须从某些专家水平的指南或手册中深入学习流缓冲相关的内容）。

 你是否发现 `Punct_stream` 的代码很易读？是否注意到注释很容易理解？你是否认为你

自己也能编写这些代码？如果你几天前还是一个真正的初学者，诚实的回答应该是“不！不！不！”甚至是“不！不！绝不可能！！你疯了吗？”我们理解你的想法，但是对于最后一个疑问，回答确实是“不，至少我们认为不可能”。我们给出本例的目的是：

- 展示一个相对实际的问题和解决方案。
- 展示用难度相对适中的方法能达到什么样的结果。
- 对于一个显然较为简单的问题，给出一个易于使用的解决方案。
- 说明接口和实现之间的差别。

为了成为一名程序员，你需要阅读真实代码，而不仅仅是仔细打磨过的用于教学的解决方案。我们给出这个例子就是出于这个目的。过了几天或几周后，这个程序对你来说就会很容易理解，你就可以考虑改进它了。

我们可以这样看待这个例子，它就相当于教师在英语初学者课上讲授了一些真正的英语俚语，为课程增加了一点色彩，活跃了学习气氛。

11.8 更多未讨论内容

I/O 相关的细节问题看上去是无穷无尽的，因为它们只受限于人类的创造力和想象力。这就如同我们无法想象自然语言有多复杂一样。英语中的 12.35 按大多数其他欧洲语言的习惯应该表示为 12,35。自然地，C++ 标准库提供了处理这一问题以及其他很多自然语言相关的 I/O 问题的功能。但是，你如何输出中文符号呢？你如何比较两个马拉雅拉姆语字符串呢？这些问题已经有解决方案了，但这些内容远远超出了本书的讨论范围。如果你对此感兴趣，请参考更为专门的或高阶的书籍（如 Langer 的《Standard C++ IOStreams and Locales》和 Stroustrup 的《The C++ Programming Language》），以及标准库和系统的文档。请搜索“本地化”（locale）一词，这个术语通常用于描述处理自然语言差异的程序设计语言特性。

另一个复杂性之源是缓冲机制：标准库 `iostream` 依赖于一个称为 `streambuf` 的机制。对于那些高端的应用（无论是从性能角度还是从功能角度），都不可避免地会用到 `streambuf`。如果你觉得需要定义自己的 `iostream`，或者需要调整 `iostream` 用于新的数据源 / 目的，参见 Stroustrup 的《The C++ Programming Language》第 38 章或者系统文档。

当使用 C++ 时，你也可能会遇到以 `printf()`/`scanf()` 为代表的 C 语言标准 I/O 函数族。如果你希望了解这部分内容，请参考 27.6 节和附录 C.10.2，或者参考 Kernighan 和 Ritchie 所编写的优秀的 C 语言教材《The C Programming Language》，以及互联网上数不清的资源。每种程序设计语言都有自己的 I/O 机制，各不相同，有的很古怪，但大多数都（以不同方式）反映了我们在第 10 章和第 11 章中所介绍的基本思想。

附录 C 中总结了 C++ 标准库的 I/O 机制。

图形用户界面（GUI）相关的话题将在第 17 ~ 21 章中进行介绍。

简单练习

1. 开始编写一个名为 `Test_output.cpp` 的程序。声明一个整数 `birth_year`，并将你的出生年份赋予它。
2. 以十进制、十六进制和八进制格式输出 `birth_year`。
3. 标出每个输出值所用的基数的名字。
4. 你是否使用制表符将输出按列对齐了？如果没有，将输出都对齐。

5. 现在输出你的年龄。
6. 现在有什么问题吗？发生什么情况了？将输出固定为十进制。
7. 返回第 2 题，让输出的每个值都显示基数。
8. 尝试读入八进制数、十六进制数，等等：

```
cin >> a >> oct >> b >> hex >> c >> d;
cout << a << '\t' << b << '\t' << c << '\t' << d << '\n';
```

运行程序，输入下面内容

```
1234 1234 1234 1234
```

解释输出结果。

9. 编写程序，分别以 `defaultfloat`、`fixed` 和 `scientific` 格式输出浮点数 1234567.89。哪种格式呈现给用户最精确的表示？并解释原因。
10. 创建一个简单的表格，包含你和至少 5 位朋友的姓、名、电话号码和电子邮件地址。试验不同的域宽，直至表格的输出满足你的需求为止。

思考题

1. 为什么 I/O 对于程序员来说比较棘手？
2. 符号 `<<hex` 的作用是什么？
3. 十六进制数在计算机科学中的作用是什么？为什么？
4. 为你想实现的几个整数输出格式化选项命名。
5. 操纵符是什么？
6. 十进制数的前缀是什么？八进制呢？十六进制呢？
7. 浮点数值的默认输出格式是哪种？
8. 什么是域？
9. 解释 `setprecision()` 和 `setw()` 的作用。
10. 文件打开模式的目的是什么？
11. 下面哪个操纵符不是持久的：`hex`、`scientific`、`setprecision()`、`showbase`、`setw`？
12. 字符串 I/O 和二进制 I/O 的差异在哪里？
13. 给出一个例子，说明使用二进制文件比使用文本文件更好。
14. 给出两个例子，说明 `stringstream` 的用途。
15. 什么是文件位置？
16. 如果你试图定位到文件尾之后，会出现什么结果？
17. 什么情况下，使用面向行的输入比面向类型的输入更好。
18. `isalnum(c)` 的功能是什么？

术语

`binary` (二进制)

`file positioning` (文件定位)

`character classification` (字符分类)

`fixed`

`decimal` (十进制)

`hexadecimal` (十六进制)

`defaultfloat`

`irregularity` (无规律)

line-oriented input (面向行的输入)
manipulator (操纵符)
nonstandard separator (非标准分隔符)
noshowbase
octal (八进制)

output formatting (输出格式)
regularity (有规律)
scientific
setprecision()
showbase

习题

- 编写程序，读取一个文本文件，将其中字母都转换为小写，生成一个新文件。
- 编写程序，给定一个文件名和一个单词，输出文件中包含该单词的每一行及其行号。提示：`getline()`。
- 编写程序，将文件中的元音都删除（“*disemvowels*”）。例如，将“Once upon a time!” 转换为“nc pn tm!”。令人惊奇的是，通常得到的结果还是可读的。请你的朋友测试这个程序。
- 编写一个名为 `multi_input.cpp` 的程序，提示用户输入几个整数，可以使用不同的数制，对八进制和十六进制分别使用 0 和 0x 前缀进行输入。程序能正确解释这些数值，并将它们转换为十进制格式。随后按列对齐输出这些数值，如下所示：

```
0x43 hexadecimal converts to 67 decimal
0123 octal      converts to 83 decimal
65 decimal      converts to 65 decimal
```

- 编写程序，读入一些字符串，对每个字符串，输出其中每个字符的分类，字符分类方式和分类函数如 11.6 节所述。注意，一个字符可能属于多个类别（如 x 既是字母又是字母数字）。
- 编写程序，将输入中的标点转换为空白符。点（.）、分号（;）、逗号（,）、问号（?）、破折号（-）、单引号（'）为标点符号，不要修改在一对双引号（"）之间的符号。例如，“– don't use the as-if rule.” 转换为“don t use the as if rule”。
- 修改上题的程序，将 don't 转换为 do not, can't 转换为 can not, 等等；在单词内的连字符保持不变（于是上题中的输入会转换为“do not use the as-if rule”）；同时将所有符号转换为小写。
- 编写程序，利用上题的程序生成字典（替代 11.7 节中的方法）。对一个多页的文本文件运行程序，观察结果是否还有改进的余地。
- 将 11.3.2 节中的二进制 I/O 程序一分为二：一个程序将原始文本文件转换为二进制，另一个程序读入二进制文件，将其转换为文本格式。测试这两个程序：对一个文本文件进行这两个步骤的转换，将结果与原始文件进行比较。
- 编写函数 `vector<string> split(const string& s)`，将 s 中以空白符分隔的子串存入向量，作为结果返回。
- 编写函数 `vector<string> split(const string& s, const string& w)`，与上题的 `split` 函数相比，新的 `split` 函数除了默认空白符外，还将 w 中的字符当作空白符。
- 编写程序，将一个文本文件中的字符颠倒顺序。例如“asdfghjkl”转换为“lkjhgfds”。
警告：反向读取文件并不是一个可移植的、高效的好方法。
- 编写程序，将一个文件中单词（空白符间隔的字符串）的顺序颠倒过来。例如，“Norwegian”

“Blue parrot”转换为“parrot Blue Norwegian”。你可以假定内存空间可以容纳文件中的所有字符串。

14. 编写程序，读取一个文本文件，输出文件中包含不同类别字符个数，字符类别定义如 11.6 节所述。
15. 编写程序，读取一个文件，文件格式是以空白符间隔的数值，将这些数值输出到另一个文件，格式采用科学记数法，精度为 8，每行 4 个域，每个域的宽度为 20 个字符。
16. 编写程序，读取一个以空白符间隔的数值文件，按升序输出这些数值，每行一个值。每个数值只输出一次，如果一个数值在原文件中出现多次，同时输出它出现的次数。例如，若原文件为“7 5 5 7 3 117 5”，则输出：

```
3
5   3
7   2
117
```

附言

 输入输出是很难处理的，因为我们人类的喜好和习惯并不遵从简单的、易于阐述的原则和直接的数学法则。作为程序员，我们几乎不可能命令用户偏离他自己的习惯和偏好；即便我们可以，最好也不要过于自大，以至于认为我们可以提供一种简单方式，来替代人类千百年来形成的习惯。因此，我们必须预计到输入输出中所面临的一定程度上的混乱，并接受它、适应它。与此同时，还要尽力保持我们程序的简洁——但不要过度简单化。

向量和自由空间

默认使用向量！

——Alex Stepanov

本章和接下来的四章将介绍 C++ 标准库（通常称为 STL）的容器和算法部分。我们介绍 STL 的关键特性及其使用。另外，介绍实现 STL 的关键设计和编程技术，以及用到的一些低层语言特性，主要包括指针、数组和自由存储空间。本章和后两章重点介绍应用最广也最有用的 STL 容器——`vector` 的设计和实现。

12.1 简介

C++ 标准库中最有用的容器就是 `vector`。一个 `vector` 提供一个指定类型元素的序列。可以通过索引（下标）引用元素，使用 `push_back()` 扩展 `vector`，使用 `size()` 获得 `vector` 元素的数量，以及对 `vector` 元素访问进行越界检查。标准库 `vector` 是一种方便、灵活、（时空）高效、静态类型安全的元素容器。标准 `string` 具有相似的特性，其他有用的标准容器类型，如 `list` 和 `map`，也是如此，我们将在第 15 章中介绍它们。但是，计算机内存并不能直接支持这些有用的类型。硬件能直接支持的只有字节序列。例如，对于一个 `vector<double>`，操作 `v.push_back(2.3)` 将 2.3 添加到 `double` 类型序列中，并将元素数量 `v(v.size())` 增加 1。在最底层，计算机并不知道 `push_back()` 这样的复杂操作的任何信息，它所知道的只是如何一次读或写若干字节。

在本章和接下来的两章中，我们将展示如何通过每个程序员都能使用的基本语言特性来构建 `vector`。通过这些内容，我们可以阐明有用的概念和编程技术，以及如何用 C++ 语言特性表达这些概念和技术。我们在 `vector` 的实现中遇到的语言特性和编程技术，都是有广泛用途且的确被广泛使用的。

在学习如何设计、实现和使用 `vector` 之后，我们可以继续学习其他的标准库容器（例如 `map`），并研究 C++ 标准库所提供的优雅、高效的特性及其使用方法（见第 15 和 16 章）。这些特性被称为算法，能将我们从涉及数据的常见编程任务中解放出来。作为替代，所有 C++ 实现都提供了许多算法给我们使用，可大大简化我们编写和测试库的工作。我们已经见到并使用过标准库中的一个最有用的算法：`sort()`。

我们通过设计一系列越来越复杂的 `vector` 实现来逐渐逼近标准库 `vector`。首先，构建一个非常简单的 `vector`。然后，考察 `vector` 中哪些部分不合需求并进行相应修改。这样经过几次处理之后，我们就得到了一个与你的 C++ 编译器所提供的标准库 `vector`（也就是在前面章节中你曾使用过的版本）大致等价的 `vector` 实现。这种逐渐完善的过程与我们完成一个新的编程任务的方式非常相似。在这个过程中，我们会遇到很多涉及内存和数据结构使用的经典问题，并对它们进行探究。我们的基本计划是：

- 第 12 章（本章）：如何处理不同大小的内存？特别是，不同 `vector` 如何拥有不同数量的元素，一个 `vector` 如何在不同时间拥有不同数量的元素？这促使我们探究自由存储空间（堆空间）、指针、类型转换（显式类型转换）和引用。
- 第 13 章：如何复制 `vector`？我们如何为它们提供下标操作？我们还会介绍数组，并讨论它与指针的关系。
- 第 14 章：如何定义不同元素类型的 `vector`？如何处理越界错误？为回答这些问题，我们将讨论 C++ 模板和异常处理特性。

除了在实现灵活、高效且类型安全的向量时介绍过的新的语言特性和技术之外，我们还会（重新）使用曾经见过的很多语言功能和编程技术。偶尔，我们会抓住机会给出一些更加正式的技术定义。

因此，在这个时刻，我们终于要直接管理内存了。为什么必须要这样做？`vector` 和 `string` 非常有用、非常方便，我们使用它们就好了。毕竟，`vector` 和 `string` 这些容器的设计目标就是让我们与实际内存处理的种种棘手问题隔离开。但是，除非我们满足于相信魔法，否则就必须探究最底层的内存管理。为什么不应“简单地相信魔法”？或者，换一个更积极的说法——为什么不应“简单地相信 `vector` 的实现者知道他们在做什么”？毕竟，我们不建议你去研究令计算机内存正常工作的物理元件是怎样的。

 好吧，我们是程序员（计算机科学家、软件开发者或其他人员），而不是物理学家。假如我们正在学习器件物理学，就必须研究计算机内存的设计细节。但是，我们正在学习程序设计，因此必须深入程序设计的细节。理论上，可以将底层内存访问和管理特性视为“实现细节”，就像我们学习器件物理学一样。但是，如果这样做，你不仅不得不“相信魔法”，还会丧失实现新容器（你会有这种需要的，这并不罕见）的能力。而且，你还会无法阅读大量直接使用内存的 C 和 C++ 代码。正如我们将在接下来的几章中所见到的，指针（一种直接引用对象的低层方法）还有很多与内存管理无关的用途。倘若不使用指针，用好 C++ 并不是件容易的事。

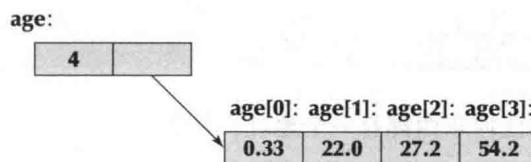
 更富哲理地讲，我和很多计算机专业人士持同样的观点：如果你对一个程序如何映射到计算机内存和操作缺乏基本的实践认识，将会在把握更高级的主题如数据结构、算法和操作系统时遇到问题。

12.2 `vector` 的基本知识

我们开始循序渐进地设计 `vector`，首先考虑一个非常简单的应用：

```
vector<double> age(4); // 一个 vector，包含 4 个 double 类型的元素
age[0]=0.33;
age[1]=22.0;
age[2]=27.2;
age[3]=54.2;
```

显然，我们创建一个有 4 个 `double` 类型元素的 `vector`，并且分别为这四个元素赋值 0.33、22.0、27.2 与 54.2。这四个元素被编号为 0、1、2 与 3。`C++` 标准库容器中的元素编号总是从 0（零）开始。从 0 开始编号是很常见的，它是 `C++` 程序员的一个通用规范。一个 `vector` 中的元素数量被称为它的大小。因此，`age` 的大小为 4。一个 `vector` 中的元素被编号（索引）为 0 到 `size()-1`。例如，`age` 中的元素被编号为 0 到 `age.size()-1`。我们可图示 `age` 如下：

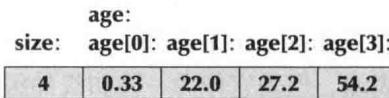


如何在计算机内存中实现这种“图解设计”？如何像这样储存和访问值呢？显然，我们需要定义一个类，并且将这个类称为 **vector**。另外，它需要一个数据成员保存它的大小，以及另一个数据成员保存它的元素。但是，如何表示一个大小可变的元素集合？可以使用标准库 **vector**，但（在此情境下）这是一种欺骗：我们正在构建一个 **vector**！

那么，如何表示上图中的箭头？先不考虑它，我们可以定义一个固定大小的数据结构：

```
class vector {
    int size, age0, age1, age2, age3;
    // ...
};
```

忽略符号表示方面的细节，我们将得到如下所示的图：



此定义简单合用，但我们第一次尝试用 **push_back()** 添加元素时就遇到了问题：无法添加元素；程序中的元素数量固定为 4 个。我们需要一个比保存固定数量元素的数据结构更强大的东西。因为如果我们将 **vector** 定义为保存固定数量的元素，那么改变元素数量的操作如 **push_back()** 就无法实现了。基本上，我们需要一个数据成员来指向一组元素，这样，当需要更大空间时可以令它指向另一组元素。这个数据成员可能是第一个元素的内存地址这样的东西。在 C++ 中，一种可以保存地址的数据类型称为指针（pointer），它在语法上使用后缀 * 来区分，因此 **double*** 表示“指向 **double** 的指针”。这样，我们就可以定义自己的第一个版本的 **vector** 类了：

```
// 一个非常简化的 double 类型 vector (类似 vector<double>)
class vector {
    int sz; // 大小
    double* elem; // 指向第一个 (double 类型) 元素的指针
public:
    vector(int s); // 构造函数：分配 double 元素
    // 令 elem 指向它们
    // 将 s 保存到 sz 中
    int size() const { return sz; } // 当前大小
};
```

在继续设计 **vector** 之前，首先详细学习“指针”的概念。“指针”概念及紧密相关的“数组”概念，是 C++ 中“内存”概念的关键部分。

12.3 内存、地址和指针

计算机的内存是一个字节序列。可以将这些字节从 0 开始编号。将这种“指明内存中位置的数字”称为地址。可以将一个地址看作一种整型值。内存中第一个字节的地址为 0，下一个字节的地址为 1，以此类推。我们可以将 1MB 字节图示如下：



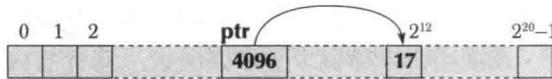
我们在内存中保存的任何东西都有一个地址。例如：

```
int var = 17;
```

这段代码为 `var` 分配一段“`int` 大小”的内存，并将值 17 保存到这段内存中。我们也可以保存地址以及操作地址。保存地址的对象被称为指针。例如，用于保存 `int` 的地址的类型称为“指向 `int` 的指针”或“`int` 指针”，其表示方法为 `int*`：

```
int* ptr = &var; // ptr 保存 var 的地址
```

“地址”运算符 `&` 用于获得一个对象的地址。因此，如果 `var` 碰巧开始于地址 4096（或 2^{12} ），`ptr` 将会保存值 4096：



基本上，我们将计算机内存看作一个字节序列，这些字节从 0 到内存大小减 1 编号。在有些计算机中这只是一种简化表示，但是它作为一种初级的编程模型，已经足够了。

每种类型都有对应的指针类型。例如：

```
int x = 17;
int* pi = &x; // int 指针

double e = 2.71828;
double* pd = &e; // double 指针
```

如果我们想查看指向的对象的值，可以使用“内容”操作符 `*`，它是一种一元运算符。例如：

```
cout << "pi==" << pi << "; contents of pi==" << *pi << "\n";
cout << "pd==" << pd << "; contents of pd==" << *pd << "\n";
```

`*pi` 的输出将是整数 17，而 `*pd` 的输出是双精度浮点数 2.71828。`pi` 和 `pd` 的输出依赖于编译器在内存中分配给变量 `x` 和 `e` 的地址。指针值（地址）的表示方式也可能不同，这依赖于你的系统所使用的规范；十六进制表示法（见附录 A.2.1.1）常用于指针值。

内容运算符（常称为解引用（dereference）运算符）也可以被用于赋值操作的左侧：

```
*pi = 27; // 正确：可以将 27 赋予 pi 指向的 int
*pd = 3.14159; // 正确：可以将 3.14159 赋予 pd 指向的 double
*pd = *pi; // 正确：可以将一个 int (*pi) 赋予一个 double (*pd)
```

需要注意的是，即使指针的值可以打印为一个整数，但是指针并不是一个整数。“一个 `int` 指向什么”不是一个好的问题；`int` 并不指向什么，只有指针才指向其他实体。指针类型提供一些适用于地址的操作，而 `int` 提供适用于整数的（算术和逻辑）操作。因此，我们不能隐式地混用指针和整数：

```
int i = pi; // 错误：不能将一个 int* 赋予一个 int
pi = 7; // 错误：不能将一个 int 赋予一个 int*
```

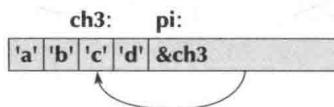
与此类似，一个指向 `char` 的指针（`char*`）不是一个指向 `int` 的指针（`int*`）。例如：

```
char* pc = pi; // 错误：不能将一个 int* 赋予一个 char*
pi = pc; // 错误：不能将一个 char* 赋予一个 int*
```

为什么将 `pc` 赋值给 `pi` 会出现错误？考虑这个答案：一个 `char` 通常比一个 `int` 更小，因此考虑下面代码：

```
char ch1 = 'a';
char ch2 = 'b';
char ch3 = 'c';
char ch4 = 'd';
int* pi = &ch3; // 指向 ch3, 一段 char 大小的内存
// 错误：我们不能将一个 char* 赋予一个 int*
// 但假装我们可以这样做
*pi = 12345; // 写入一段 int 大小的内存
*pi = 67890;
```

编译器如何在内存中分配变量是由具体 C++ 实现定义的，但是我们很可能得到如下内容：



现在，假如这段代码编译通过，我们可以将 12345 写入从 `&ch3` 开始的内存。这肯定会改变相邻内存中的值，例如 `ch2` 或 `ch4`。如果我们真的不走运（可能性很高），就会覆盖 `pi` 本身的部分！在此情况下，下次赋值 `*pi=67890` 会将 67890 放入内存中完全不同的地方。令人高兴的是这种赋值是不允许的，这是编译器为这种低层编程提供的少数几种保护之一。

在极少数情况下，你可能真的需要把一个 `int` 转换成一个指针，或者将一个指针类型转换成另一种指针类型，此时可使用 `reinterpret_cast`，见 12.8 节。

在这些例子中我们真的很接近硬件了，这对于程序员来说并不是特别舒服的地方。因为只有很少几种原语操作可供我们使用，并且几乎得不到语言或标准库的支持。但是，我们只有接近硬件层次，才能了解 `vector` 这样的高层特性是如何实现的。我们需要了解如何在这个层次编写代码，因为并不是所有代码都是“高层”代码（见第 25 章）。而且，只有在缺少高层软件的便利性和相对安全性的条件下编写过代码，我们才能更深刻地理解它们的重要性。我们的目标是：在给定了问题和求解约束的条件下，永远在尽可能高的抽象层次上进行程序开发。在本章和第 13、14 章中，我们通过 `vector` 的实现来介绍如何回到一个更舒服的抽象层次上编写代码。

12.3.1 `sizeof` 运算符

那么，一个 `int` 实际占用多少内存？一个指针呢？运算符 `sizeof()` 可以回答这些问题：

```
void sizes(char ch, int i, int* p)
{
    cout << "the size of char is " << sizeof(char) << ' ' << sizeof(ch) << '\n';
    cout << "the size of int is " << sizeof(int) << ' ' << sizeof(i) << '\n';
    cout << "the size of int* is " << sizeof(int*) << ' ' << sizeof(p) << '\n';
}
```

正如你所看到的，我们可以将 `sizeof` 用于一个类型名或表达式。对一个类型名，`sizeof` 给出这种类型的对象的大小；对一个表达式，`sizeof` 给出表达式结果的大小。`sizeof` 的结果

是一个正整数，其单位是 `sizeof(char)`——被定义为 1。一个 `char` 通常被保存在一个字节中，因此 `sizeof` 会报告占用的字节数。

试一试

执行上面这个例子，观察我们能得到什么。然后，扩展这个例子以确定 `bool`、`double` 和其他某些类型的大小。

一种类型的大小并不保证在所有 C++ 实现中都相同。目前，`sizeof(int)` 在台式机或笔记本电脑中通常为 4 字节。如果使用 8 比特的字节，那就意味着一个 `int` 是 32 比特。但是，嵌入式系统通常使用 16 比特的 `int`，而高性能体系结构中常使用 64 比特的 `int`。

一个 `vector` 使用多少内存？我们可以尝试下面代码：

```
vector<int> v(1000);      // 1000 个 int 类型的元素的 vector
cout << "the size of vector<int>(1000) is " << sizeof(v) << '\n';
```

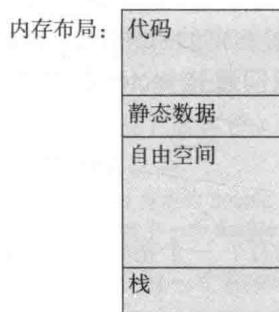
输出可能像下面这样：

```
the size of vector<int>(1000) is 20
```

经过本章和下一章（见 14.2.1 节）的学习，将很容易解释为什么得到这样的输出，但目前我们至少知道，`sizeof` 显然不是统计元素数量。

12.4 自由空间和指针

思考一下 12.2 节结尾的 `vector` 实现。`vector` 从哪里获得它的元素的空间？我们如何令指针 `elem` 指向它们？当你开始编写一个 C++ 程序时，编译器为你的代码分配内存（有时称为代码存储（code storage）或文本存储（text storage）），并为你定义的全局变量分配内存（称为静态存储（static storage））。编译器还会为你预留调用函数时所需的空间，函数需要用这些空间保存其参数和局部变量（称为栈存储（stack storage）或自动存储（automatic storage））。计算机中的剩余内存可用于其他用途；它是“自由的”（“空闲的”）。这种内存分配方式可图示如下：

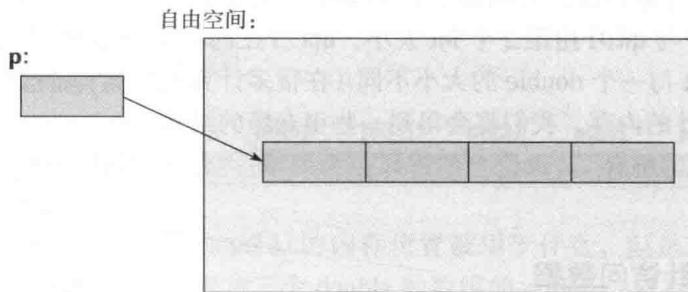


C++ 语言用称为 `new` 的运算符将“自由空间”（free store，又称为堆（heap））变为可用状态。例如：

```
double* p = new double[4];      // 在自由空间中分配 4 个 double
```

这段代码要求 C++ 运行时系统在自由空间中分配四个 `double`，并将指向第一个 `double`

的指针返回给我们。我们使用此指针来初始化指针变量 p。可图示如下：



运算符 new 返回一个指向它创建的对象的指针。如果它创建了多个对象（一个数组），它返回指向第一个对象的指针。如果对象的类型是 X，则 new 返回的指针类型是 X*。例如：

```
char* q = new double[4]; // 错误：将 double* 赋予 char*
```

如果 new 返回一个指向 double 的指针，而一个 double 并不是一个 char，因此我们不应（也不能）将它赋予 char 指针变量 q。

12.4.1 自由空间分配

我们使用运算符 new 来请求系统从自由空间中分配（allocate）内存：

- 运算符 new 返回一个指向分配的内存的指针。
- 指针的值是分配的内存的首字节的地址。
- 一个指针指向一个特定类型的对象。
- 一个指针并不知道它指向多少个元素。

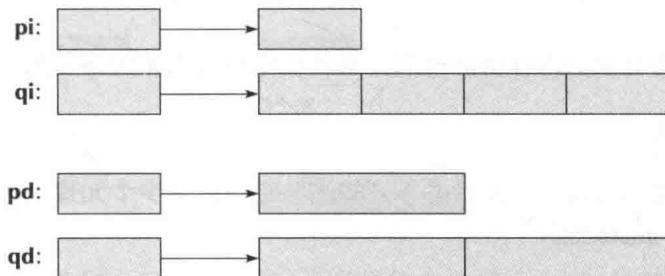


运算符 new 可以为单个元素分配内存，也可为元素序列（数组）分配内存。例如：

```
int* pi = new int; // 分配一个 int
int* qi = new int[4]; // 分配 4 个 int (一个包含 4 个 int 的数组)
```

```
double* pd = new double; // 分配一个 double
double* qd = new double[n]; // 分配 n 个 double (一个包含 n 个 double 的数组)
```

注意，分配的对象数量可以通过一个变量给出。这很重要，因为这样我们就可以在运行时选择分配多少个对象。如果 n 等于 2，我们得到：



指向不同类型变量的指针是不同类型。例如：

```
pi = pd; // 错误：不能将一个 double* 赋予一个 int*
pd = pi; // 错误：不能将一个 int* 赋予一个 double*
```

为什么不可以？毕竟，我们可以将一个 `int` 赋给一个 `double`，以及将一个 `double` 赋给一个 `int`。原因在于 `[]` 运算符。它依赖于元素类型的大小来计算出到哪里找到一个元素。例如，`qi[2]` 在内存中与 `qi[0]` 相距 2 个 `int` 大小，`qd[2]` 在内存中与 `qd[0]` 有 2 个 `double` 大小的距离。如果一个 `int` 与一个 `double` 的大小不同（在很多计算机中确实是这样），那么如果允许将 `qi` 指向分配给 `qd` 的内存，我们将会得到一些很奇怪的结果。

这是“实践上的解释”。理论上的解释更为简单：“允许为指针分配不同类型将会导致类型错误”。

12.4.2 通过指针访问数据

在一个指针上除了使用解引用运算符 `*` 之外，我们还可以使用下标运算符 `[]`。例如：

```
double* p = new double[4]; // 在自由空间中分配 4 个 double
double x = *p;           // 读取 p 指向的（第一个）对象
double y = p[2];          // 读取 p 指向的第三个对象
```

不出所料，下标运算符与 `vector` 的下标运算符一样都是从 0 开始计数，因此 `p[2]` 引用第三个元素；`p[0]` 是第一个元素，因此 `p[0]` 实际上与 `*p` 相同。`[]` 和 `*` 运算符也可以被用于写入数据：

```
*p = 7.7;                // 写入 p 指向的（第一个）对象
p[2] = 9.9;               // 写入 p 指向的第三个对象
```

一个指针指向内存中的一个对象。“内容”运算符（又称为解引用运算符）允许我们读取或写入指针 `p` 指向的对象：

```
double x = *p;           // 读取 p 指向的对象
*p = 8.8;                 // 写入 p 指向的对象
```

当我们将 `[]` 运算符作用于一个指针时，它将内存看作一个对象（类型在指针声明时指定）序列，指针 `p` 指向其中第一个对象：

```
double x = p[3];          // 读取 p 指向的第四个元素
p[3] = 4.4;                // 写入 p 指向的第四个元素
double y = p[0];            // p[0] 与 *p 等价
```

这就是全部。这里没有越界检查和巧妙的实现，只是简单地访问我们计算机的内存：

<code>p[0]:</code>	<code>p[1]:</code>	<code>p[2]:</code>	<code>p[3]:</code>
8.8		9.9	4.4

这种简单但最为有效的内存访问机制正是我们实现 `vector` 所需要的。

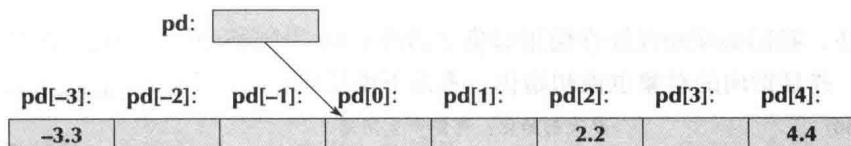
12.4.3 指针范围

⚠ 指针带来的主要问题是一个指针并不“知道”它指向多少个元素。考虑下面代码：

```
double* pd = new double[3];
pd[2] = 2.2;
pd[4] = 4.4;
pd[-3] = -3.3;
```

`pd` 是否有第三个元素 `pd[2]`？它是否有第五个元素 `pd[4]`？如果查看 `pd` 的定义，我们发现答案分别是“是”和“否”。但是，编译器不知道这些；它并不跟踪指针的值。这段代

码只是简单地访问内存，就像我们已经分配的足够的内存一样。它甚至会访问 `pd[-3]`，就像  `pd` 指向的地址往前三个 `double` 的位置也是我们分配的内存的一部分一样：



我们并不知道标记为 `pd[-3]` 和 `pd[4]` 的内存位置被用于什么。但是，即使我们知道也不意味着它们可以作为 `pd` 指向的包含三个 `double` 的数组的一部分。最有可能的情况是，它们是其他对象的一部分，而我们将它们弄得乱七八糟。这不是一个好主意。实际上，这是一个典型的灾难性的坏主意：“灾难性”表现在“我的程序神秘崩溃”或“我的程序得到错误的输出”。尝试着大声说出来；它听起来根本不好。我们要走很长一段路来避免这种错误。越界访问特别令人讨厌，因为程序中明显无关的部分会受到影响。一次越界的读取会给我们一个“随机”值，它可能依赖于某些完全无关的计算。一次越界的写入会将某些对象变成“不可能”的状态，或者简单地赋予它一个不期望的错误值。这种写入通常在发生后很长时间内都不会被注意到，因此很难被发现。更糟糕的是：你运行一个带有越界错误的程序两次，输入稍有不同就可能出现不同的结果。这种错误（“瞬时错误”）是最难发现的错误之一。

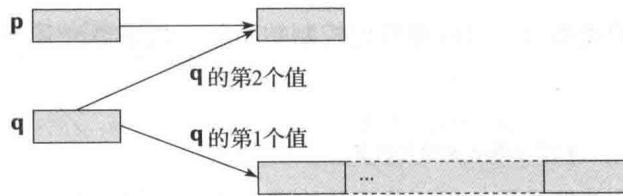
我们必须保证不出现这种越界的访问。我们使用 `vector` 而不是直接使用 `new` 来分配内存的原因之一是 `vector` 知道自己的大小，这样它（或我们）就很容易避免越界的访问。

有一个因素令避免越界访问变得困难，那就是我们可以将一个 `double*` 赋予另一个 `double*`，而不必去管每个指针指向多少个对象。一个指针实际上并不知道它指向多少个对象。例如：

```
double* p = new double;           // 分配一个 double
double* q = new double[1000];      // 分配 1000 个 double

q[700] = 7.7;                     // 很好
q = p;                            // 令 q 与 p 指向相同地址
double d = q[700];                // 越界访问!
```

仅仅在 3 行代码中，`q[700]` 引用了两个不同的内存位置，第二次是越界的访问，并且很可能引发一场灾难。



现在，我们希望你会问：“为什么指针不会记住自己的大小？”很显然，我们可以设计一个能记住大小的“指针”，`vector` 差不多就是如此。如果你阅读 C++ 文献和库，你将会发现很多“智能指针”可以弥补内置低层指针的缺陷。但是，有时我们需要接触硬件层次并理解对象如何寻址——一个机器地址并不“知道”地址中是什么内容。而且，理解指针才能理

解大量的实际代码。

12.4.4 初始化

一如以往，我们必须要保证在使用对象之前为它赋一个值；也就是说，我们希望确保指针被初始化，并且指向的对象也被初始化。考虑下面代码：

```
double* p0;           // 未初始化：可能产生问题
double* p1 = new double; // 得到（分配）一个未初始化的 double
double* p2 = new double{5.5}; // 得到一个初始化为 5.5 的 double
double* p3 = new double[5]; // 得到（分配）5 个未初始化的 double
```

显然，声明 p0 但没有对它进行初始化会带来麻烦。考虑下面代码：

```
*p0 = 7.0;
```

这行代码将 7.0 赋给内存中的某个位置，但我们并不知道将会是哪部分内存。这个赋值可能是无害的，但是永远也不要这样做。我们迟早会得到与越界访问相同的结果：“我的程序神秘崩溃”或“我的程序得到错误的输出”。对于老式 C++ 程序（“C 风格程序”），

 大多数严重错误是由未初始化指针的访问或越界的访问而引起。我们必须尽最大努力去避免这种访问，部分原因是着眼于专业化，部分原因是我们不想浪费时间查找这种错误。很少有事情像查找这种错误一样令人沮丧和厌倦。避免错误而不是查找它更令人愉快和有效率。

 对于内置类型，使用 new 分配的内存不会被初始化。如果想初始化单个对象，你可指定一个值，就像我们对 p2 所做的：*p2=5.5。注意 {} 初始化语法。它与 [] 相对，后者表示“数组”。

对 new 分配的对象数组，我们可以指定一个初始化器列表。例如：

```
double* p4 = new double[5] {0,1,2,3,4};
double* p5 = new double[] {0,1,2,3,4};
```

现在，p4 指向 5 个 double 类型的变量，它们的值为 0.0、1.0、2.0、3.0 和 4.0。p5 也是如此；如果提供了一组元素作为初始值，我们可以省略元素数目。

像往常一样，我们应该小心未初始化的对象，确保在读取它们之前已为它们赋值。注意，编译器通常有一个“调试模式”，每个变量被默认初始化为一个预期值（通常为 0）。这意味着当关闭调试模式并发布一个程序时、当用优化器优化程序时或者只是在不同机器上编译时，带有未初始化变量的程序可能突然有不同的运行结果。不要陷入未初始化变量的麻烦中。

当我们定义自己的类型时，可以更好地控制初始化。如果类型 X 有一个默认构造函数，我们会得到：

```
X* px1 = new X;           // 一个默认初始化的 X
X* px2 = new X[17];        // 17 个默认初始化的 X
```

如果类型 Y 有一个构造函数，但不是默认构造函数，我们需要显式地初始化：

```
Y* py1 = new Y;           // 错误：无默认构造函数
Y* py2 = new Y{13};        // 正确：初始化为 Y{13}
Y* py3 = new Y[17];        // 错误：无默认构造函数
Y* py4 = new Y[17] {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
```

为 new 指定很长的初始化器列表可能不太实用，但当我们只需要几个元素时这种方式

就非常方便了，而少量元素通常是最常见的情形。

12.4.5 空指针

如果你没有其他指针用来初始化一个指针，那么使用空指针 `nullptr`:

```
double* p0 = nullptr; // 空指针
```

当 0 值被赋给一个指针时，它被称为空指针（null pointer）。我们经常通过检测指针是否为 `nullptr` 来检查一个指针是否有效（它是否指向什么东西）。例如：

```
if (p0 != nullptr) // 认为 p0 有效
```

这并不是一个完美的检测，因为 `p0` 可能包含一个碰巧不是 0 的“随机”值（例如我们忘记了初始化），或者一个已经被删除对象的地址（见 12.4.6 节）。但是，一般来说这已是我们所能做得最好的。我们实际上不必明确提及 `nullptr`，因为 `if` 语句会检测它的条件是否为 `nullptr`：

```
if (p0) // 认为 p0 有效；等价于 p0!=nullptr
```

考虑到它更直接地表达了“`p0` 有效”的思想，我们更喜欢这种简短的形式，但是也有不同意见。

如果我们有一个指针有时指向一个对象，而有时什么都不指向，我们就需要使用空指针。这种情况很少见，比很多人想象的更少。思考一下：如果你没有对象需用一个指针指向，那你又为什么定义那个指针呢？你不能等到有一个对象时再做吗？

用名字 `nullptr` 表示空指针是 C++11 的新特性，因此在旧代码中，人们通常使用 0（零）或 `NULL` 代替 `nullptr`。两种旧的替代方法都会导致混淆和 / 或错误，因此优先选择更专用的 `nullptr`。

12.4.6 自由空间释放

`new` 运算符会从自由空间中分配（“获取”）内存。由于一台计算机的内存是有限的，因此在使用完毕后将内存归还到自由空间通常是一个好主意。这样自由空间可以将这些内存重新用于新的分配请求。对于大型程序和长时间运行的程序来说，这种自由空间的重用是很重要的。例如：

```
double* calc(int res_size, int max) // 内存泄漏
{
    double* p = new double[max];
    double* res = new double[res_size];
    // 用 p 计算结果，放入 res
    return res;
}

double* r = calc(100,1000);
```

就像注释中所写的那样，每次调用 `calc()` 会导致分配给 `p` 的 `double` “泄漏”。例如，调用 `calc(100,1000)` 会导致 1000 个 `double` 占用的空间在程序接下来的运行过程中无法使用。

将内存归还自由空间的运算符称为 `delete`。我们对 `new` 返回的指针使用 `delete`，令这些内存重新变为自由空间中的可用内存，可供未来分配。现在，这个例子变为：

```

double* calc(int res_size, int max)
    // 调用者负责释放为 res 分配的内存
{
    double* p = new double[max];
    double* res = new double[res_size];
    // 用 p 计算结果，放入 res
    delete[] p; // 我们不再需要这段内存，将其释放
    return res;
}

double* r = calc(100,1000);
// use r
delete[] r; // 我们不再需要这段内存，将其释放

```

这个例子还顺带说明了使用自由内存的一个主要原因：我们可以在一个函数中创建对象，并将它们传回调用者。

delete 有两种形式：

- **delete p** 释放 **new** 分配给单个对象的内存。
- **delete[] p** 释放 **new** 分配给对象数组的内存。

虽然有些烦人，但选择正确的形式是程序员的责任。



两次删除一个对象是一个糟糕的错误。例如：

```

int* p = new int{5};
delete p; // 很好：p 指向由 new 创建的对象
// ... no use of p here ...
delete p; // 错误：p 指向的内存是由自由空间管理器所拥有的

```

第二个 **delete p** 带来两个问题：

- 你已不再拥有指针指向的对象，因此自由空间管理器可能已经改变了它的内部数据结构，导致你无法再次正确地执行 **delete p**。
- 自由空间管理可能已“回收”**p** 指向的内存，因此 **p** 现在可能指向其他对象；删除这个对象（由程序的其他部分所拥有）会引起错误。

这两个问题都发生在实际的程序中；而不仅仅是存在理论上的可能性。

删除空指针不会做任何事（因为空指针不指向一个对象），因此删除空指针是无害的。例如：

```

int* p = nullptr;
delete p; // 很好：不需要任何动作
delete p; // 仍然很好（仍然不需要任何动作）

```

为什么我们要被内存释放问题所困扰？编译器不能指出我们什么时候不再需要一段内存，并在没有人工干预的情况下将它回收吗？它可以，这被称为自动垃圾收集（automatic garbage collection）或垃圾收集（garbage collection）。不幸的是，自动垃圾收集是有代价的，而且并不是对所有应用都是理想的。如果你确实需要自动垃圾收集，可以将一个自动垃圾收集器插入你的 C++ 程序。你可以找到一些好的垃圾收集器（见 www.stroustrup.com/C++.html）。但是，本书假设你必须处理自己的“垃圾”，我们会教你如何便捷、高效地完成这项工作。



什么时候内存不泄漏是重要的？一个需要“永远”运行的程序是无法忍受内存泄漏的。操作系统就是“永远运行的”程序的例子，大多数的嵌入式系统（见第 25 章）也属于此类。

库也不能出现内存泄漏的问题，因为有人可能将它用作不能有内存泄漏的系统的一部分。一般来说，简单地对所有程序都保证不泄漏是个好主意。很多程序员将泄漏的原因归结于马虎。但这并没有切中要点。当你在某种操作系统（Unix、Windows等）上运行一个程序，程序结束时所有内存都将会自动归还给系统。结果就是，如果你知道自己的程序使用的内存量不比系统可用内存量更多，你可能“合理地”决定泄漏内存，直到操作系统为你释放内存。但是，如果你决定这样做，先确认你所估计的内存消耗是正确的，否则人们有理由认为你是草率的。

12.5 析构函数

现在，我们知道如何为一个 `vector` 保存元素。我们简单地为这些元素在自由空间中分配足够的空间，并且通过一个指针来访问它们：

```
// 一个高度简化的 double 的 vector
class vector {
    int sz;                                // 大小
    double* elem;                           // 指向元素的指针
public:
    vector(int s)                         // 构造函数
        :sz(s),                            // 初始化 sz
        elem(new double[s])                // 初始化 elem
    {
        for (int i=0; i<s; ++i) elem[i]=0; // 初始化元素
    }
    int size() const { return sz; }         // 当前大小
    ...
};
```

这样，`sz` 就是元素的数量。我们在构造函数中初始化它，用户可以通过调用 `size()` 得到 `vector` 中的元素数量。在构造函数中使用 `new` 来分配元素空间，从自由空间返回的指针被保存在成员指针 `elem` 中。

注意，我们将元素初始化为它们的默认值（0.0）。标准库 `vector` 就是这样做的，因此我们认为在一开始最好也这样做。

不幸的是，我们第一个 `vector` 版本会泄漏内存。在构造函数中，它使用 `new` 来为元素分配内存。遵循在 12.4 节中描述的规则，我们必须保证使用 `delete` 释放这些内存。思考下面程序：

```
void f(int n)
{
    vector v(n);           // 分配 n 个 double
    ...
}
```

当我们离开函数 `f()` 时，`v` 在自由空间中创建的元素没有释放。我们可以为 `vector` 定义一个 `clean_up()` 操作并调用它：

```
void f2(int n)
{
    vector v(n);           // 定义一个 vector (分配另外 n 个 int)
    ... use v ...
    v.clean_up();          // clean_up() 释放元素
}
```

这段代码运行良好。但是，自由空间一个最常见的问题是人们会忘记 `delete`。`clean_up()` 也会产生同样问题，人们可能忘记调用它。我们可以做得更好。基本思路是令编译器知道一个可以做与构造函数相反事情的函数，就像它了解构造函数一样。必然地，这个函数被称为析构函数（destructor）。就像一个类对象创建时会隐式调用构造函数一样，当一个对象离开其作用域时会隐式调用析构函数。构造函数确保一个对象被正确创建并初始化。与之相反，析构函数确保一个对象销毁前被正确清理。例如：

```
// 一个高度简化的 double 的 vector
class vector {
    int sz;                                // 大小
    double* elem;                           // 指向元素的指针
public:
    vector(int s)                          // 构造函数
        :sz{s}, elem{new double[s]}        // 分配内存
    {
        for (int i=0; i<s; ++i) elem[i]=0; // 初始化元素
    }

    ~vector()                            // 析构函数
    { delete[] elem; }                  // 释放内存
    // ...
};
```

有了这个定义，我们可以编写下面的代码：

```
void f3(int n)
{
    double* p = new double[n];           // 分配 n 个 double
    vector v(n);                        // 分配 n 个 double 的 vector
    // 使用 p 和 v
    delete[] p;                         // 释放 p 的内存
} // v 离开作用域后 vector 自动进行清理工作
```

突然之间，`delete[]` 看起来相当令人厌烦且容易出错。有了 `vector`，我们其实就没有理由使用 `new` 分配内存，而在函数结束时使用 `delete[]` 释放内存了。这些工作 `vector` 都能完成，而且做得更好。特别是，`vector` 不会忘记调用它的析构函数释放元素使用的内存。

我们在这里并不打算介绍析构函数使用的更多细节，但是对于那些需要首先（从某处）获取并随后归还的资源，如文件、线程、锁等，析构函数是很好的管理机制。回忆一下 `iostream` 如何在使用完毕后进行清理工作。它们刷新缓冲区、关闭文件、释放缓冲区空间等。这些都是由它们的析构函数完成的。每个“拥有”资源的类都需要一个析构函数。

12.5.1 生成的析构函数

如果一个类的成员拥有一个析构函数，则在包含这个成员的对象销毁时这个析构函数会被调用。例如：

```
struct Customer {
    string name;
    vector<string> addresses;
    // ...
};

void some_fct()
{
```

```

Customer fred;
// 初始化 fred
// 使用 fred
}

```

当我们退出函数 `some_fct()` 时，`fred` 将会离开其作用域，因此 `fred` 会被销毁；也就是说，`name` 和 `addresses` 的析构函数会被调用。显然，只有这样析构函数才是有用的，它有时被表达为“编译器为 `Customer` 生成一个析构函数，它调用成员的析构函数”。这种通常的实现方式正是析构函数调用应提供的显然且必要的保证。

成员和基类的析构函数被派生类的析构函数（无论是用户定义的或是编译器生成的）隐式调用。基本上，所有规则可总结为一句话：“当对象被销毁时，析构函数被调用（离开作用域、被 `delete` 释放等情况下）。”

12.5.2 析构函数和自由空间

析构函数概念上很简单，但它是大多数最有效的 C++ 编程技术的基础。基本思想是：

- 无论一个类对象需要使用哪些资源，这些资源都要在构造函数中获取。
- 在对象的生命周期中，它可以释放资源并获得新的资源。
- 在对象的生命周期结束后，析构函数释放对象拥有的所有资源。

在 `vector` 中用一对构造函数、析构函数处理自由空间内存就是这一思想的一个典型例子。我们将在 14.5 节中提供这种思想的更多例子。本节将考察一个重要的应用，它结合使用了自由空间和类层次。考虑下面代码：

```

Shape* fct()
{
    Text tt {Point{200,200}, "Annemarie"};
    // ...
    Shape* p = new Text{Point{100,100}, "Nicholas"};
    return p;
}

void f()
{
    Shape* q = fct();
    // ...
    delete q;
}

```

这段代码看起来很合理，事实也的确如此，它运行一切正常。不过让我们分析一下它是如何做到的，这段代码揭示了一些精巧、重要和简单的技术。在函数 `fct()` 中，`Text`（见 18.11 节）对象 `tt` 在离开函数时被正确销毁。`Text` 有一个 `string` 成员，很明显需要调用它的析构函数——`string` 像 `vector` 一样处理内存的获取和释放。对于 `tt`，正确销毁是很容易的；编译器只需像 12.5.1 节中描述的那样调用 `Text` 的生成析构函数即可。但是，从 `fct()` 返回的 `Text` 对象会怎样？调用者 `f()` 完全不了解 `q` 指向一个 `Text`，它所知道的是 `q` 指向一个 `Shape`。`delete q` 如何调用 `Text` 的析构函数？

在 19.2.1 节中，我们快速地介绍了 `Shape` 的析构函数。实际上，`Shape` 的析构函数是 `virtual` 的，这就是问题的关键。当我们使用 `delete q` 时，`delete` 会查看 `q` 的类型，以确定是否需要调用析构函数，如果需要的话就调用它。因此，`delete q` 会调用 `Shape` 的析构函数 `~Shape()`。但是，`~Shape()` 是 `virtual` 的，因此会使用 `virtual` 调用机制（见 19.3.1 节）——这

个调用会转向 Shape 的派生类的析构函数，在本例中是 ~Text()。如果 Shape::~Shape() 不是虚函数，Text::~Text() 将不会被调用，Text 的 string 成员也就不会被正确销毁。

作为一个经验法则：如果你有一个类带有 virtual 函数，则它也需要一个 virtual 析构函数。原因是：

- 如果一个类有 virtual 函数，它很可能作为一个基类使用；
- 且如果它是一个基类，它的派生类很可能使用 new 来分配；
- 且如果派生类对象使用 new 来分配，并通过基类指针来操作；
- 那么它很可能也是通过基类指针来进行 delete 操作的。

注意，析构函数是通过 delete 来隐式或间接调用的，不会直接调用，这样能省去很多麻烦的工作。

试一试

编写一个使用基类和成员的小程序，为基类和成员定义构造函数和析构函数，在调用时输出一行信息。然后，创建几个对象并观察它们的构造函数和析构函数如何被调用。

12.6 访问元素

为了令 vector 可用，我们需要一种读写元素的方法。我们第一步可以先提供简单的 get() 和 set() 成员函数：

```
// 一个高度简化的 double 的 vector
class vector {
    int sz;                                // 大小
    double* elem;                           // 指向元素的指针
public:
    vector(int s) :sz{s}, elem{new double[s]} {/* ... */} // 构造函数
    ~vector() { delete[] elem; }             // 析构函数

    int size() const { return sz; }          // 当前大小

    double get(int n) const { return elem[n]; } // 访问：读
    void set(int n, double v) { elem[n]=v; }   // 访问：写
};
```

get() 和 set() 都可以对 elem 指针使用 [] 运算符来访问元素：elem[n]。

现在，我们可以创建一个 double 的 vector 并使用它：

```
vector v(5);
for (int i=0; i<v.size(); ++i) {
    v.set(i, 1.1*i);
    cout << "v[" << i << "]==" << v.get(i) << '\n';
}
```

这段代码将会输出

```
v[0]==0
v[1]==1.1
v[2]==2.2
v[3]==3.3
v[4]==4.4
```

这仍是一个过于简单的 `vector`，而且使用 `get()` 和 `set()` 的代码相对于常用的下标语法也很粗陋。但是，我们的目的是从简单的小程序开始，逐步扩展我们的程序，并一直进行测试。这种扩展和反复测试的策略可以减少错误和调试工作量。

12.7 指向类对象的指针

“指针”的概念是通用的，因此我们可以指向可放置于内存的任何东西。例如，我们可以使用指向 `vector` 的指针，就像使用指向 `char` 的指针一样：

```
vector* f(int s)
{
    vector* p = new vector(s); // 在自由空间中分配一个 vector
    // 填充 *p
    return p;
}

void ff()
{
    vector* q = f(4);
    // 使用 *q
    delete q; // 在自由空间中释放 vector
}
```

注意，当我们 `delete` 一个 `vector` 时，它的析构函数会被调用。例如：

```
vector* p = new vector(s); // 在自由空间中分配一个 vector
delete p; // 释放
```

当在自由空间中创建一个 `vector` 时，`new` 运算符：

- 首先为 `vector` 分配内存。
- 然后，调用 `vector` 的构造函数来初始化 `vector`；构造函数为 `vector` 的元素分配内存，并初始化这些元素。

当删除 `vector` 时，`delete` 运算符：

- 首先调用 `vector` 的析构函数；这个析构函数调用元素的析构函数（如果它们有析构函数），然后释放元素使用的内存。
- 然后，释放 `vector` 使用的内存。

注意，这个过程是如何完美地递归执行的（见 8.5.8 节）。如果使用实际的（标准库）`vector`，我们还可以实现：

```
vector<vector<double>>* p = new vector<vector<double>>(10);
delete p;
```

这里，`delete p` 调用 `vector<vector<double>>` 的析构函数；接着，这个析构函数调用它的 `vector<double>` 元素的析构函数，所有东西都被干净利落地清理，不会留下未销毁的对象和泄漏的内存。

由于 `delete`（为具有析构函数的类型，例如 `vector`）调用析构函数，我们通常称它是在销毁对象，而不只是释放它们。

照例，请记住一个位于构造函数之外的“裸”`new` 很可能导致我们忘记 `delete` 由 `new` 创建的对象。除非你有一个删除对象的好（即，真的非常简单，例如 18.10 节和附录 E.4 中的 `Vector_ref`）的策略，否则尽量将 `new` 放在构造函数中，将 `delete` 放在析构函数中。

到目前为止一切都很好，但是如果仅仅给定一个指针，我们如何访问一个 `vector` 的成员？注意，所有类都支持对给定的对象名通过“.”（点）运算符来访问其成员：

```
vector v(4);
int x = v.size();
double d = v.get(3);
```

与此类似，所有类都支持对给定的对象指针通过`->`（箭头）运算符来访问其成员：

```
vector* p = new vector(4);
int x = p->size();
double d = p->get(3);
```

类似`.`（点），`->`（箭头）既可用于数据成员也可用于函数成员。由于内置类型（例如 `int` 和 `double`）没有成员，因此`->`不能用于内置类型。点和箭头通常被称为成员访问运算符。

12.8 类型混用：`void*` 和类型转换

在使用指针和自由空间分配的数组时，我们非常接近硬件层面。基本上，我们对指针的操作（初始化、分配、`*` 和`[]`）直接映射为机器指令。在这个层次，语言只提供一点儿表示上的便利以及由类型系统提供的编译时的一致性。偶尔，我们不得不放弃这最后一点儿保护。

通常，我们不希望在没有类型系统的保护下工作，但是有时没有其他选择（例如，我们需要与无法识别 C++ 类型的其他语言交互）。我们有时也会遇到一些不走运的情况，这时需要面对没有遵循静态安全类型设计的旧代码。在这种情况下，我们需要两样东西：

- 一种并不了解内存中是哪种对象的指针。
- 一个操作，告知编译器指针指向的内存中是哪种（未证实）类型的对象。

 类型 `void*` 的含义是“指向编译器不知道类型的内存空间”。当我们想在两段代码之间传输一个地址，它们确实不知道对方的类型时，就可以使用 `void*`。这方面的例子包括回调函数的“地址”参数（见 21.3.1 节）和底层内存分配器（例如 `new` 运算符的实现）。

并不存在 `void` 类型的对象，但是正如我们看到的，我们通常用 `void` 来表示“没有返回值”：

```
void v; // 错误：不存在 void 类型的对象
void f(); // f() 不返回任何东西——f() 不是返回一个 void 类型的对象
```

指向任何对象类型的指针都可以赋予 `void*`。例如：

```
void* pv1 = new int; // 正确：int* 转换为 void*
void* pv2 = new double[10]; // 正确：double* 转换为 void*
```

由于编译器不知道 `void*` 指向什么，因此我们必须告诉它：

```
void f(void* pv)
{
    void* pv2 = pv; // 正确拷贝（void* 是可以进行拷贝的）
    double* pd = pv; // 错误：不能将 void* 转换为 double*
    *pv = 7; // 错误：不能解引用一个 void*
              // (我们不知道它指向的对象是什么类型)
    pv[2] = 9; // 错误：不能对 void* 进行下标操作
    int* pi = static_cast<int*>(pv); // 正确：显式类型转换
    ...
}
```

 `static_cast` 可以用于在两种相关指针类型之间进行强制转换，例如 `void*` 与 `double*`（见

附录 A.5.7)。名字 `static_cast` 是故意为一个丑陋的(且危险的)操作起的一个丑陋的名字, 你只应在绝对必要时才使用它。你经常会发现其实没有必要使用。`static_cast` 这样的操作称为显式类型转换(explicit type conversion, 因为这正是它所做的事)或口语化地称为转换(`cast`, 由于它用来支持被打破的东西)。

C++ 提供两个比 `static_cast` 更具潜在危险的转换操作:

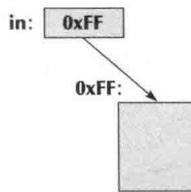
- `reinterpret_cast` 可以在两个不相关的类型之间转换, 例如 `int` 和 `double`。
- `const_cast` 可以“去掉 `const`”。

例如:

```
Register* in = reinterpret_cast<Register*>(0xff);
```

```
void f(const Buffer* p)
{
    Buffer* b = const_cast<Buffer*>(p);
    ...
}
```

第一个例子是有关 `reinterpret_cast` 的必要性和使用方法的经典例子。我们告知编译器内存中的某个特定部分(开始于 `0xFF` 位置的内存)被作为一个 `Register`(可能有特殊语义)。在你编写设备驱动程序这类代码时, 采用这种方法是必要的。



在第二个例子中, `const_cast` 将 `const` 从名为 `p` 的 `const Buffer*` 中去掉。我们想必知道自己在做什么。

`static_cast` 至少不会混淆指针/整数的区别或出现“`const` 问题”, 因此在你感到有必要使用显式类型转换时最好用 `static_cast`。当你认为需要进行转换时, 重新考虑: 是否有办法不进行转换就能编写出代码? 是否有办法重新设计部分程序从而不再需要转换? 除非你需要与硬件或其他人的代码打交道, 否则通常都会有办法避免使用转换。否则, 你将会面对微妙和讨厌的错误。你不要期望使用 `reinterpret_cast` 的代码可以移植。

12.9 指针和引用

我们可以将一个引用看作一个自动解引用的不可变指针或一个对象的替代名字。指针和引用有以下几方面不同:

- 为一个指针赋值会改变指针自身的值(不是指针指向的值)。X
- 为了得到一个指针, 你通常需要使用 `new` 或 `&`。
- 为了访问一个指针指向的对象, 你可以使用 `*` 或 `[]`。
- 为一个引用赋值会改变引用指向的值(不是引用自身的值)。
- 在初始化一个引用之后, 你不能让引用指向其他对象。
- 为引用赋值执行深拷贝(赋值给引用的对象); 为指针赋值不是这样(赋值给指针自身)。

- 注意避免空指针。

例如：

```
int x = 10;
int* p = &x;           // 你需要 & 来获取指针
*p = 7;               // 用 * 通过 p 为 x 赋值
int x2 = *p;           // 通过 p 读取 x
int* p2 = &x2;         // 获取另一个 int 的指针
p2 = p;               // p2 和 p 都指向 x
p = &x2;               // 令 p 指向另一个对象
```

对应的引用的例子如下：

```
int y = 10;
int& r = y;           // & 在类型中，不是在初始化器中
r = 7;                // 通过 r 为 y 赋值（不需要 *）
int y2 = r;           // 通过 r 读取 y（不需要 *）
int& r2 = y2;         // 获取另一个 int 的引用
r2 = r;               // 将 y 的值赋予 y2
r = &y2;               // 错误：你不能改变引用自身的值
// (不能将一个 int* 赋予一个 int&)
```

注意最后一个例子；不仅是这种语言构造会失败，而是一个引用在初始化后就无法再指向其他对象了。如果你需要指向不同的对象，请使用指针。如需了解如何使用指针，请参见 12.9.3 节。

引用与指针都是通过使用内存地址来实现的。它们只是在地址的使用上不同，为编程人员提供稍有不同的特性。

12.9.1 指针参数和引用参数

当你希望将一个变量的值改为由函数计算出的结果时，你有三种选择。例如：

```
int incr_v(int x) { return x+1; }    // 计算一个新值并返回
void incr_p(int* p) { ++*p; }        // 传递一个指针
                                         // (解引用它并递增)
void incr_r(int& r) { ++r; }        // 传递一个引用
```

你会如何选择呢？我们认为返回值是最明显的方法（因此最不容易出错），例如：

```
int x = 2;
x = incr_v(x);           // 将 x 拷贝到 incr_v()，然后将结果拷贝出来并赋予 x
```

我们倾向于对小对象（例如一个 int）使用这种风格。此外，如果一个“大对象”有移动构造函数（见 13.3.4 节），我们也可以高效地反复拷贝它。

我们如何选择使用引用参数还是指针参数呢？不幸的是，两者都有自己的吸引力和问题，因此没有明确的答案。你需要根据具体函数及其可能的用途来决定。

使用指针参数警告编程者有些东西可能改变。例如：

```
int x = 7;
incr_p(&x)           // 需要 &
incr_r(x);
```

在 incr_p(&x) 中需要使用 &，这警告用户 x 可能改变。与之对比，incr_p(x) 就是“看起来无辜的”。这导致很多人稍微偏爱使用指针。

 另一方面，如果你使用指针作为函数的参数，需要小心某些人可能用空指针（即 nullptr）

调用函数。例如：

```
incr_p(nullptr);           // 崩溃：incr_p() 会尝试解引用空指针
int* p = nullptr;
incr_p(p);                // 崩溃：incr_p() 会尝试解引用空指针
```

这显然很讨厌。编写 incr_p() 的人可以防止它：

```
void incr_p(int* p)
{
    if (p==nullptr) error("null pointer argument to incr_p()");
    ++*p;      // 解引用指针并递增指向的对象
}
```

但是现在 incr_p() 不像以前看起来那么简单、有吸引力了。第 5 章讨论如何处理糟糕的参数。与之相比，引用的用户（例如 incr_r()）有权假设引用的确指向一个对象。

如果“不传递任何东西”（不传递对象）从函数语义的角度是可接受的，那么我们就必须使用指针参数。注意：递增操作并不是这种情况，因此它需要当 `p==nullptr` 时抛出一个异常。

因此，实际的答案是：“如何选择依赖于函数的性质。”具体如下：

- 对于小对象，优先使用传值参数。
- 对于“没有对象”（用 `nullptr` 表示）是有效参数的函数，使用指针参数（记住检测 `nullptr`）。
- 否则，使用一个引用参数。

参见 8.5.6 节。

12.9.2 指针、引用和继承

在 19.3 节中，我们看到 Circle 这样的派生类如何被用在需要其公有基类 Shape 的对象的地方。我们可以用指针或引用来表达这个思想：由于 Shape 是 Circle 的一个公有基类，因此 `Circle*` 可以被隐式转换为 `Shape*`。例如：

```
void rotate(Shape* s, int n);           // 将 *s 旋转 n 度

Shape* p = new Circle{Point{100,100},40};
Circle c{Point{200,200},50};
rotate(p,35);
rotate(&c,45);
```

对于引用是类似的：

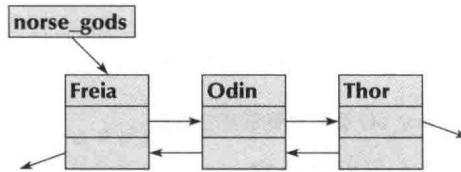
```
void rotate(Shape& s, int n);           // 将 s 旋转 n 度

Shape& r = c;
rotate(r,55);
rotate(*p,65);
rotate(c,75);
```

这是大多数的面向对象的编程技术的关键（见 19.3 ~ 19.4 节）。

12.9.3 实例：链表

链表是最普通、最有用的数据结构之一。列表通常由一些“链接”组成，每个链接会保存一些信息和指向其他链接的指针。这是指针的典型用途之一。例如，一个挪威众神的短链表可图示如下：



一个像这样的链表被称为双向链表（doubly-linked list），对每个链接，我们都可以获得其前驱与后继。一个只能找到后继的列表被称为单向链表（singly-linked list）。当我们希望很容易地移除一个元素时，应使用双向链表。我们可以定义链接如下：

```
struct Link {
    string value;
    Link* prev;
    Link* succ;
    Link(const string& v, Link* p = nullptr, Link* s = nullptr)
        : value{v}, prev{p}, succ{s} {}
};
```

如果获得一个链接，我们可以使用 `succ` 指针到达它的后继，或者使用 `prev` 指针到达它的前驱。我们使用空指针来表示一个没有后继或前驱的链接。我们可以构造自己的挪威众神链表如下：

```
Link* norse_gods = new Link("Thor",nullptr,nullptr);
norse_gods = new Link("Odin",nullptr,norse_gods);
norse_gods->succ->prev = norse_gods;
norse_gods = new Link("Freia",nullptr,norse_gods);
norse_gods->succ->prev = norse_gods;
```

我们构建链表的方式是创建多个 `Link`，并如图中所示将它们连接起来：首先是 Thor，然后是 Odin，它是 Thor 的前驱，最后是 Freia，它是 Odin 的前驱。你可以沿着指针查看，我们构建了正确的链表，每个后继和前驱都指向正确的神。但是，这段代码有些混乱晦涩，因为我们没有明确地定义和命名插入操作。

```
Link* insert(Link* p, Link* n) // 在 p 之前插入 n (不完整)
{
    n->succ = p;           // p 紧跟在 n 之后
    p->prev->succ = n;   // n 紧跟在 p 的前驱之后
    n->prev = p->prev;    // p 的前驱变为 n 的前驱
    p->prev = n;          // n 变为 p 的前驱
    return n;
}
```

如果 `p` 真的指向一个 `Link`，且 `p` 指向的 `Link` 真的有一个前驱，这个函数会正确运行。请说服你自己确实是这样。当我们思考指针与链接结构（例如由 `Link` 组成的链表）时，我们总是在纸上画出小框和箭头组成的图示来验证我们的代码对小例子能正确运行。但请不要太过于依赖这种有效但没什么技术含量的设计技术。

这个版本的 `insert()` 是不完整的，因为它没有处理 `n`、`p` 或 `p->prev` 为 `nullptr` 的情况。我们增加适当的空指针测试，得到一个稍乱但正确的版本：

```
Link* insert(Link* p, Link* n) // 在 p 之前插入 n; 返回 n
{
    if (n==nullptr) return p;
    if (p==nullptr) return n;
    n->succ = p;           // p 紧跟在 n 之后
```

```

if (p->prev) p->prev->succ = n;
n->prev = p->prev;      // p 的前驱变为 n 的前驱
p->prev = n;            // n 变为 p 的前驱
return n;
}

```

有了这个定义，我们可以编写代码如下

```

Link* norse_gods = new Link("Thor");
norse_gods = insert(norse_gods,new Link("Odin"));
norse_gods = insert(norse_gods,new Link("Freia"));

```

现在所有易于出错的 `prev` 和 `succ` 指针操作都从我们的视线中消失了。指针操作烦人且易错，而且可能会隐藏在编写与测试好的函数中。特别是，传统代码中的很多错误是由于程序员忘记测试指针是否为 `nullptr`，正如我们（故意）在 `insert()` 的第一个版本所做的那样。

注意，我们使用了默认参数（见 20.3.1 节和附录 A.9.2），使得用户不必每次使用构造函数时都要指出前驱与后继。

12.9.4 链表操作

在标准库中提供了一个 `list` 类，我们将会在 15.4 节中介绍它。`list` 类隐藏了所有链接操作，但在本节中我们将基于 `Link` 类详细介绍 `list` 的概念，以便对 `list` 类“隐藏在表面之下”的运行机制有一些感觉，并学习更多指针使用的例子。

我们的 `Link` 类需要哪些操作来令用户避免“摆弄指针”？这在某种程度上是一个人偏好问题，但下面这组操作通常是有用的：

- 构造函数。
- `insert`: 在一个元素前插入。
- `add`: 在一个元素后插入。
- `erase`: 删除一个元素。
- `find`: 查找保存给定值的 `Link`。
- `advance`: 获得第 n 个后继。

我们可以像下面这样实现这些操作：

```

Link* add(Link* p, Link* n)    // 在 p 之后插入 n; 返回 n
{
    // 很像 insert (见习题 11)
}

Link* erase(Link* p)           // 从链表中删除 *p; 返回 p 的后继
{
    if (p==nullptr) return nullptr;
    if (p->succ) p->succ->prev = p->prev;
    if (p->prev) p->prev->succ = p->succ;
    return p->succ;
}

Link* find(Link* p, const string& s)    // 在链表中查找 s
                                         // 返回 nullptr 表示“未找到”
{
    while (p) {
        if (p->value == s) return p;
        p = p->succ;
    }
}

```

```

    return nullptr;
}

Link* advance(Link* p, int n) // 在链表中移动 n 个位置
// 返回 nullptr 表示“未找到”
// n 为正数表示前进，负数表示后退
{
    if (p==nullptr) return nullptr;
    if (0<n) {
        while (n--) {
            if (p->succ == nullptr) return nullptr;
            p = p->succ;
        }
    }
    else if (n<0) {
        while (n++) {
            if (p->prev == nullptr) return nullptr;
            p = p->prev;
        }
    }
    return p;
}

```

注意后缀 `n++` 的使用。这种形式的递增（“后递增”）的结果是递增前的旧值。

12.9.5 链表的使用

作为一个小的练习，我们建立两个链表：

```

Link* norse_gods = new Link("Thor");
norse_gods = insert(norse_gods,new Link{"Odin"});
norse_gods = insert(norse_gods,new Link{"Zeus"});
norse_gods = insert(norse_gods,new Link{"Freia"});

Link* greek_gods = new Link("Hera");
greek_gods = insert(greek_gods,new Link{"Athena"});
greek_gods = insert(greek_gods,new Link{"Mars"});
greek_gods = insert(greek_gods,new Link{"Poseidon"});

```

不幸的是，我们犯了两个错误：Zeus 是一位希腊的天神，而不是一位北欧的天神；希腊的战争之神是 Ares，而不是 Mars（Mars 是他的拉丁 / 罗马名字）。我们可以修改它：

```

Link* p = find(greek_gods, "Mars");
if (p) p->value = "Ares";

```

注意，我们是如何小心处理 `find()` 返回 `nullptr` 的情况的。我们认为在本例中这种情况下不可能发生（毕竟我们刚刚将 Mars 插入 `greek_gods` 中），但在实际的例子中某些人可能改变代码。

与之相似，我们可以将 Zeus 移入正确的神殿中：

```

Link* p = find(norse_gods, "Zeus");
if (p) {
    erase(p);
    insert(greek_gods,p);
}

```

你是否注意到了错误？它对你来说应该是一个相当微妙的错误（除非你已熟悉直接处理链接）。如果我们用 `erase()` 删除的恰好是 `norse_gods` 指向的 `Link` 会怎样呢？在这段代码中

这种情况同样不会发生，但是为了编写可维护的优质代码，我们必须考虑这种可能性。

```
Link* p = find(norse_gods, "Zeus");
if (p) {
    if (p==norse_gods) norse_gods = p->succ;
    erase(p);
    greek_gods = insert(greek_gods,p);
}
```

当我们修改第一个错误时，同时也修改了第二个错误：当我们在第一个希腊天神之前插入 Zeus 时，我们需要让 greek_gods 指向 Zeus 的 Link。指针非常有用、非常灵活，但也是很微妙的。

最后，让我们打印出这个链表：

```
void print_all(Link* p)
{
    cout << "{ ";
    while (p) {
        cout << p->value;
        if (p=p->succ) cout << ", ";
    }
    cout << "}";
}

print_all(norse_gods);
cout << "\n";

print_all(greek_gods);
cout << "\n";
```

将会输出：

```
{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

12.10 this 指针

注意，在我们的每个链表函数中，都使用一个 `Link*` 作为第一个参数，并访问这个对象中的数据。我们把这种函数定义为成员函数。我们是否可以通过将操作变为成员函数来简化 `Link`（或链接的使用）呢？我们是否可能令指针私有，以便只有成员函数能够访问它们呢？这些都是可以办到的：

```
class Link {
public:
    string value;

    Link(const string& v, Link* p = nullptr, Link* s = nullptr)
        : value(v), prev(p), succ(s) {}

    Link* insert(Link* n);           // 在此对象之前插入 n
    Link* add(Link* n);            // 在此对象之后插入 n
    Link* erase();                 // 将此对象从链表中删除
    Link* find(const string& s);   // 在链表中查找 s
    const Link* find(const string& s) const; // 在 const 链表中查找 s (见 18.5.1 节)

    Link* advance(int n) const;     // 在链表中移动 n 个位置

    Link* next() const { return succ; }
    Link* previous() const { return prev; }}
```

```
private:
    Link* prev;
    Link* succ;
};
```

这看起来是很有前途的方法。对于不改变 Link 状态的操作，我们将它们定义为 const 成员函数。我们增加了（非修改性）函数 next() 和 previous()，从而用户可以遍历链表（的 Link）——这两个函数现在是必要的，因为直接访问 succ 和 prev 被禁止了。我们仍将 value 定义为一个公有成员，因为（到目前为止）我们还没有理由不这样做——它“只是数据”。

现在尝试实现 Link::insert()，我们可以赋值我们之前编写的全局函数 insert() 并适当修改它：

```
Link* Link::insert(Link* n)      // 在 p 之前插入 n; 返回 n
{
    Link* p = this;           // 指向此对象的指针
    if (n==nullptr) return p; // 未插入任何东西
    if (p==nullptr) return n; // 原链表为空
    n->succ = p;            // p 紧跟在 n 之后
    if (p->prev) p->prev->succ = n;
    n->prev = p->prev;       // p 的前驱变为 n 的前驱
    p->prev = n;             // n 变为 p 的前驱
    return n;
}
```



但我们如何获得调用 Link::insert() 的那个对象的指针呢？如果没有语言的帮助，我们是办不到的。但是，在每个成员函数中，标识符 this 都是指向调用此成员函数的对象的指针。我们可以简单地使用 this 而不再使用 p：

```
Link* Link::insert(Link* n)      // 在本对象之前插入 n; 返回 n
{
    if (n==nullptr) return this;
    if (this==nullptr) return n;
    n->succ = this;          // 本对象紧跟在 n 之后
    if (this->prev) this->prev->succ = n;
    n->prev = this->prev;     // 本对象的前驱
                               // 变为 n 的前驱
    this->prev = n;           // n 变为本对象的前驱
    return n;
}
```

这有点儿啰嗦，但当访问成员时我们其实不必提及 this，因此代码可简化为：

```
Link* Link::insert(Link* n) // 在本对象之前插入 n; 返回 n
{
    if (n==nullptr) return this;
    if (this==nullptr) return n;
    n->succ = this;          // 本对象紧跟在 n 之后
    if (prev) prev->succ = n;
    n->prev = prev;           // 本对象的前驱变为 n 的前驱
    prev = n;                 // n 变为本对象的前驱
    return n;
}
```

换句话说，每当我们访问一个成员时，都隐式使用了 this 指针——指向当前对象的指针。只有当我们要涉及整体对象时才需要显式指出它。

注意，this 具有特殊含义：它指向调用当前成员函数的对象。它不指向任何旧对象。编

译器确保我们不能在成员函数中改变 this 的值。例如：

```
struct S {
    // ...
    void mutate(S* p)
    {
        this = p;    // 错误：this 不可变
        // ...
    }
};
```

12.10.1 关于链表使用的更多讨论

在处理实现的细节之后，我们可以看到链表的使用变成如下形式：

```
Link* norse_gods = new Link("Thor");
norse_gods = norse_gods->insert(new Link("Odin"));
norse_gods = norse_gods->insert(new Link("Zeus"));
norse_gods = norse_gods->insert(new Link("Freia"));

Link* greek_gods = new Link("Hera");
greek_gods = greek_gods->insert(new Link("Athena"));
greek_gods = greek_gods->insert(new Link("Mars"));
greek_gods = greek_gods->insert(new Link("Poseidon"));
```

它与之前的版本非常相似。与前面的例子一样，我们改正自己的“错误”。修改战争之神的名字：

```
Link* p = greek_gods->find("Mars");
if (p) p->value = "Ares";
```

将 Zeus 移到正确的神殿：

```
Link* p2 = norse_gods->find("Zeus");
if (p2) {
    if (p2==norse_gods) norse_gods = p2->next();
    p2->erase();
    greek_gods = greek_gods->insert(p2);
}
```

最后，打印出这个链表：

```
void print_all(Link* p)
{
    cout << "{ ";
    while (p) {
        cout << p->value;
        if (p=p->next()) cout << ", ";
    }
    cout << " }";
}

print_all(norse_gods);
cout << "\n";

print_all(greek_gods);
cout << "\n";
```

将会输出：

```
{ Freia, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

你更喜欢哪个版本：`insert()` 等操作是成员函数的版本，还是它们是独立函数的版本？对于本例，两者的差别并不大，但请参考 9.7.5 节。

通过观察发现，我们仍没有一个链表类，只有一个链接类。这样我们就必须一直为哪个指针指向第一个元素而操心。我们可以通过定义一个 `List` 类来做得更好，但沿着本节的路线进行设计是很常见的。我们将在 15.4 节中介绍标准库 `list`。

简单练习

本章的简单练习包括两部分。第一部分练习建立对自由空间数组的理解，并与 `vector` 进行比较：

1. 使用 `new` 分配一个由 10 个 `int` 组成的数组。
2. 使用 `cout` 打印这 10 个 `int` 的值。
3. 使用 `delete[]` 释放这个数组。
4. 编写一个函数 `print_array10(ostream& os, int* a)`，将 `a`（假设包含 10 个元素）的值打印到 `os`。
5. 分配一个由 10 个 `int` 组成的数组；用值 100、101、102 等初始化数组；打印数组的值。
6. 分配一个由 11 个 `int` 组成的数组；用值 100、101、102 等初始化数组；打印数组的值。
7. 编写一个函数 `print_array(ostream& os, int* a, int n)`，将 `a`（假设包含 `n` 个元素）的值打印到 `os`。
8. 分配一个由 20 个 `int` 组成的数组；用值 100、101、102 等初始化数组；打印数组的值。
9. 你是否记得删除这个数组？（如果没有，删除它。）
10. 重复执行第 5、6、8 步，使用一个 `vector` 来代替数组，使用一个 `print_vector()` 来代替 `print_array()`。

第二部分集中在指针及指针与数组的关系上。使用来自上一个练习的 `print_array()`：

1. 分配一个 `int`，将它初始化为 7，并将它的地址分配给变量 `p1`。
2. 打印 `p1` 的值和它指向的 `int` 的值。
3. 分配一个由 7 个 `int` 组成的数组；将它初始化为 1、2、4、8 等；将它的地址分配给变量 `p2`。
4. 打印 `p2` 的值和它指向的数组的值。
5. 声明一个名字为 `p3` 的 `int*`，并使用 `p2` 来初始化它。
6. 将 `p1` 赋值给 `p2`。
7. 将 `p3` 赋值给 `p2`。
8. 打印 `p1` 和 `p2` 的值和它们指向的数组的值。
9. 释放所有从自由空间分配的内存。
10. 分配一个由 10 个 `int` 组成的数组；将它初始化为 1、2、4、8 等；将它的地址分配给变量 `p1`。
11. 分配一个由 10 个 `int` 组成的数组，并将它的地址赋值给变量 `p2`。
12. 将由 `p1` 指向的数组的值复制到由 `p2` 指向的数组。
13. 重复第 10 至 12 步，使用一个 `vector` 来代替数组。

思考题

1. 为什么我们需要元素数量可变的数据结构？

2. 一个典型的程序包含哪四类存储?
3. 什么是自由空间? 它常用的其他名称是什么? 哪种运算符支持它?
4. 什么是解引用运算符, 为什么我们需要它?
5. 地址是什么? 在 C++ 中如何操纵内存地址?
6. 指针中包含指向的对象的什么信息? 又缺少什么有用的信息?
7. 一个指针可以指向什么?
8. 什么是泄漏?
9. 什么是资源?
10. 我们如何初始化一个指针?
11. 什么是空指针? 我们什么时候需要使用它?
12. 我们什么时候需要一个指针 (而不是一个引用或具名对象)?
13. 什么是析构函数? 我们什么时候需要使用它?
14. 我们什么时候需要一个 `virtual` 析构函数?
15. 成员的析构函数如何被调用?
16. 什么是转换? 我们什么时候需要使用它?
17. 我们如何通过指针访问类成员?
18. 什么是双向链表?
19. 什么是 `this`? 我们什么时候需要使用它?

术语

<code>address</code> (地址)	<code>member destructor</code> (成员析构函数)
<code>address of: &</code> (地址: <code>&</code>)	<code>memory</code> (内存)
<code>allocation</code> (分配)	<code>memory leak</code> (内存泄漏)
<code>cast</code> (转换)	<code>new</code>
<code>container</code> (容器)	<code>null pointer</code> (空指针)
<code>contents of: *</code> (内容: <code>*</code>)	<code>nullptr</code>
<code>deallocation</code> (释放)	<code>pointer</code> (指针)
<code>delete</code>	<code>range</code> (指针范围)
<code>delete[]</code>	<code>resource leak</code> (资源泄漏)
<code>dereference</code> (解除引用)	<code>subscripting</code> (下标操作)
<code>destructor</code> (析构函数)	<code>subscript: []</code> (下标: <code>[]</code>)
<code>free store</code> (自由空间)	<code>this</code>
<code>link</code> (链接)	<code>type conversion</code> (类型转换)
<code>list</code> (链表)	<code>virtual destructor</code> (<code>virtual</code> 析构函数)
<code>member access: -></code> (成员访问: <code>-></code>)	<code>void*</code>

习题

1. 你使用的 C++ 编译器中, 指针值的输出形式是怎样的? 提示: 不要查阅你的 C++ 编译器文档。
2. 一个 `int` 占多少字节? `double` 呢? `bool` 呢? 不要使用 `sizeof`, 除非你要验证自己的答案。

3. 编写一个函数 `void to_low(char* s)`, 将 C 风格字符串 `s` 中的所有大写字符都替换为对应的小写字符。例如, “Hello, World!” 替换为 “hello, world!”。不要使用任何标准库函数。C 风格字符串是一个由 0 结束的字符数组, 因此在结尾你会发现一个值为 0 的 `char`。
4. 编写一个函数 `char* strdup(const char*)`, 将 C 风格字符串复制到自由空间上分配的内存中。不要使用任何标准库函数。
5. 编写一个函数 `char* findx(const char* s, const char* x)`, 在 C 风格字符串 `s` 中查找字符串 `x` 首次出现的位置。
6. 本章没有说明当你使用 `new` 用尽内存时会发生什么。这种情况被称为内存耗尽 (memory exhaustion)。搞清楚将会发生什么。你有两个明显的选择: 查阅你的 C++ 编译器的文档, 或者编写一个用无限循环分配内存但不释放的程序。两者都尝试一下。在失败之前你大约分配了多少内存?
7. 编写一个程序从 `cin` 读取字符保存到自由空间上分配的数组中。读取字符直到输入感叹号 (!) 为止。不要使用 `std::string`。不要担心内存耗尽。
8. 重新做练习 7, 但这次将读取的字符存入 `std::string`, 而不是自由空间上的内存中 (`string` 知道如何使用自由空间)。
9. 栈向哪个方向生长: 向上 (趋向高地址) 还是向下 (趋向低地址)? 自由空间初始时向哪个方向生长 (在使用 `delete` 之前)? 编写程序来找到答案。
10. 查看你对练习 7 的解决方案。是否有办法输入数组而导致溢出; 也就是说, 是否能输入比为数组分配的空间更多的字符 (一个严重的错误)? 如果你试图输入比分配的空间更多的字符时, 将会发生什么合理的现象?
11. 完成 12.10.1 中的“众神链表”例子并运行它。
12. 为什么我们要定义两个版本的 `find()`?
13. 修改 12.10.1 中的 `Link` 类以保存 `struct God` 的值。`struct God` 包含 `string` 类型的成员: 姓名、神话体系、坐骑和武器。例如, `God("Zeus", "Greek", "", "lightning")` 和 `God("Odin", "Norse", "Eight-legged flying horse called Sleipner", "")`。编写一个函数 `print_all()`, 每行列出一位天神的属性。添加一个成员函数 `add_ordered()`, 将 `new` 元素按字典序放置在正确的位置。使用保存 `God` 类型值的 `Link`, 构建来自三个神话体系的天神列表; 然后将元素 (天神) 从这个链表移到三个字典序排列的链表中, 每个链表对应于一个神话体系。
14. 是否可以使用单向链表来实现 12.10.1 节中的“众神列表”例子; 即, 我们是否可以将 `prev` 成员排除在 `Link` 之外? 为什么我们希望这样做? 单向链表对哪种例子有意义? 只使用单向链表来重新实现这个例子。

附言

当我们可以简单地使用 `vector` 时, 为什么要困扰于指针和自由空间这样杂乱的、低层次的东西呢? 一个答案是有些人设计和实现了 `vector` 以及类似的抽象, 而我们希望知道它是如何工作的。有些编程语言并不提供相当于指针的功能, 因此就将这个问题留给低层的编程。基本上, 这些语言的编程者将对硬件的直接访问任务交给 C++ 编程者 (或其他适于低层编程的语言的编程者)。但我们最喜欢的理由其实更简单——你只有看到了软件是如何适应硬件的, 才能真正宣称自己了解计算机和编程。那些不知道指针、内存地址等的人, 对于编程语言特性是如何工作的经常会有很奇怪的想法; 这种错误的想法会催生“有趣的糟糕”代码。

向量和数组

购者自慎。

——忠告

本章阐述如何拷贝 `vector` 以及如何通过下标操作访问 `vector`。为此，我们将讨论一般的拷贝技术并讨论 `vector` 与数组这一低层概念之间的关系。在另一方面，本章将阐述数组与指针之间的关系以及它们使用中的一些问题。本章还会介绍对任何类型都必不可少的五种操作：构造、默认构造、拷贝构造、拷贝赋值以及析构。此外，容器类型还需要移动构造运算符和移动赋值操作。

13.1 简介

为了能够在蓝天中翱翔，一架飞机首先需要在跑道上加速直至它的速度足以摆脱地球的引力。当飞机在跑道上隆隆移动时，它不过就是一辆特别笨重的、丑陋的大卡车；而一旦飞上了天空，它就会立刻变成一种完全不同的优雅的、高效的交通工具——这才是它的真正本领。

在本章中，我们处于“奔跑”的途中，我们的最终目的是学习足够的编程语言特性与编程技术，以便能摆脱直接管理计算机内存所带来的困难和束缚。我们希望达到这样一种境界：我们只需用一些类型进行编程，这些类型就完全具备逻辑需求所希望的特性。为了“抵达那里”，我们首先需要解决很多与裸机访问有关的基本限制，例如：

- 一个对象在内存中的大小是固定的。
- 一个对象存放在内存中的某一特定位置。
- 计算机只为这些对象提供了有限的基本操作（如拷贝一个机器字、将两个字的值相加，等等）。

基本上，这些都属于 C++ 内置类型与操作的限制（从 C 语言继承而来的硬件上的限制；参见 22.2.5 节与第 27 章）。在第 12 章中，我们已经开始设计 `vector` 类型，它能控制对其元素的所有访问，并且提供了一些从用户角度看来（而非从硬件角度看）“很自然”的操作。

本章将着重介绍拷贝这一概念。这是一个重要的技术问题：我们对一个复杂对象的拷贝所希望的语义是什么？拷贝之后主体与副本之间的独立程度如何？有几种拷贝操作？我们如何指定使用哪种拷贝操作？拷贝操作与其他基本操作（例如初始化与清理）之间有什么关系？

当不能使用高层类型（如 `vector` 与 `string`）时，我们不可避免地需要讨论如何直接操纵内存。我们将讨论数组和指针以及它们之间的关系、它们的用法和使用中容易犯的错误。这些信息对每一个开始接触 C++ 或 C 低层编程的人都是十分重要的。

我们在学习的过程中应留意 `vector` 类型针对 `vector` 对象的特有细节以及 C++ 在低层类型基础上构建高层类型的方法。然而，各种语言中的“高层”类型（`string`、`vector`、`list`、`map` 等）是基于相同的机器原语构建的，反映了本章所讨论问题的不同解法。

13.2 初始化

考虑我们在第 12 章结束时定义的 `vector`:

```
class vector {
    int sz;           // 大小
    double* elem;    // 指向元素的指针
public:
    vector(int s)      // 构造函数
        :sz{s}, elem{new double[s]} /* ... */ // 分配内存
    ~vector()          // 析构函数
        { delete[] elem; }           // 释放内存
    ...
};
```

这个定义很好，但如果我们希望用一组值初始化向量元素，而不是将它们初始化为默认值，会怎样呢？例如：

```
vector v1 = {1.2, 7.89, 12.34};
```

我们可以这样做，这比将元素初始化为默认值然后再将我们希望的值赋予它们的方式好得多：

```
vector v2(2);      // 冗长易错
v2[0] = 1.2;
v2[1] = 7.89;
v2[2] = 12.34;
```

与 `v1` 的初始化相比，`v2` 的初始化冗长易错（我们在这段代码中就故意将元素数量弄错了）。如使用 `push_back()`，我们就不必提及 `vector` 的大小：

```
vector v3;          // 冗长重复
v2.push_back(1.2);
v2.push_back(7.89);
v2.push_back(12.34);
```

但这种方法仍会产生很多重复代码，那么我们如何编写接受初始化器列表参数的构造函数呢？用 {} 限定的类型 T 元素的列表是以标准库类型 `initializer_list<T>` 对象（即 T 的列表）的形式呈现给程序员的，因此我们可以编写如下代码：

```
class vector {
    int sz;           // 大小
    double* elem;    // 指向元素的指针
public:
    vector(int s)      // 构造函数 (s 为元素数量)
        :sz{s}, elem{new double[s]} // 为元素分配未初始化的内存
    {
        for (int i = 0; i < sz; ++i) elem[i] = 0.0; // 初始化
    }

    vector(initializer_list<double> lst)      // 初始化器列表构造函数
        :sz{lst.size()}, elem{new double[sz]} // 为元素分配未初始化的内存
    {
        copy(lst.begin(), lst.end(), elem); // 初始化 (用 std::copy(); 见附录 C.5.2 )
    }
    ...
};
```

我们使用了标准库算法 `copy`（见附录 C.5.2）。它将前两个参数（在本例中是 `initializer_list` 的起始和结束位置）指定的元素序列拷贝到从第三个参数开始的元素序列（在本例中是

从开始 elem 的 vector 的元素) 中。

现在我们可以编写如下代码:

```
vector v1 = {1,2,3}; // 三个元素 1.0、2.0、3.0
vector v2(3); // 三个元素，都具有(默认值) 0.0
```

注意，我们是如何用()表示元素数量，以及用{}表示元素列表的。我们需要从表示形式上将它们区分开来。例如：

```
vector v1{3}; // 一个元素，值为 3.0
vector v2(3); // 三个元素，都具有(默认值) 0.0
```

这种区分方式不很优雅，但很有效。如果有多个构造函数可供选择，编译器会将{}列表中的一个值解释为一个元素值，并将它作为一个 initializer_list 的元素传递给初始化器列表构造函数。

在大多数情况下（包括我们将在本书中遇到的所有情况），{} 初始化器列表前的=是可选的，因此我们可以编写如下代码：

```
vector v11 = {1,2,3}; // 三个元素 1.0、2.0、3.0
vector v12{1,2,3}; // 三个元素 1.0、2.0、3.0
```

两个定义只是风格上的不同。

注意，我们是按传值方式传递 initializer_list<double> 的。我们是故意这样做的，这也是语言规则所要求的：一个 initializer_list 只是指向分配在“别处的”元素的句柄（见附录 C.6.4）。

13.3 拷贝

再次考虑我们的不完整的 vector：

```
class vector {
    int sz; // 大小
    double* elem; // 指向元素的指针
public:
    vector(int s) // 构造函数
        :sz{s}, elem{new double[s]} /* ... */ // 分配内存
    ~vector() // 析构函数
        { delete[] elem; } // 释放内存
    ...
};
```

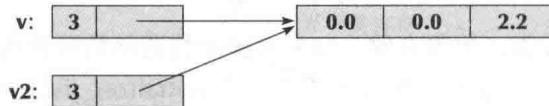
让我们尝试拷贝一个向量：

```
void f(int n)
{
    vector v(3); // 定义一个包含 3 个元素的 vector
    v.set(2,2.2); // 将 v[2] 设置为 2.2
    vector v2 = v; // 会发生什么?
    ...
}
```

理想情况下，v2 变成 v 的一个副本（即 = 意味着拷贝）；也就是说，v2.size() == v.size()，且对于所有位于区间 [0: v.size()) 内的整数 i 有 v2[i] == v[i]。并且，对象 v2 与 v 所占用的内存将在函数 f() 结束时被系统回收。这些正是标准库 vector 所做的（当然是这样），但我们的非常简单的 vector 还不能实现这些。我们的任务是完善 vector 使之能够正确处理这个例子，

但在此之前，我们首先需要弄清目前的版本都能做什么，准确地说是哪里做错了？如何做错的？为什么？一旦找出了错误所在，我们就很可能解决这些错误。更重要的，当在其他场景中遇到类似问题时，我们就能意识到问题并加以避免。

 对一种类型而言，拷贝的默认含义是“拷贝所有的数据成员”。这通常完全讲得通。例如，我们拷贝一个 `Point` 对象就是要拷贝其坐标。但对于指针成员而言，仅仅对指针成员进行拷贝会产生问题。特别是，以我们例子中的 `vector` 对象为例，这种默认语义意味着拷贝完成后 `v.sz == v2.sz` 且 `v.elem == v2.elem`，这样我们的 `vector` 对象会像下面这样：



也就是说，`v2` 并未拥有 `v` 的元素的副本；它只是共享了 `v` 的元素。我们可以写出如下代码：

```

v.set(1,99);           // set v[1] to 99
v2.set(0,88);          // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);

```

输出结果将会是 88 99，这不是我们想要的结果。假如 `v` 与 `v2` 之间没有这种“隐藏的”关联，我们将会得到输出 0 0，因为我们根本没有向 `v[0]` 或 `v2[1]` 写入值。你可能会认为当前版本的行为是“有趣的”、“简洁的”或“有时是有用的”，但这不是我们想要的，也不是标准库 `vector` 所提供的。并且，我们从 `f()` 返回时发生的事情肯定会导致一场灾难：`v` 与 `v2` 的析构函数被隐式调用；`v` 的析构函数会通过如下语句释放元素所占用的内存：

```
delete[] elem;
```

而之后 `v2` 的析构函数也会这样做。由于 `v` 与 `v2` 的 `elem` 指向同一块内存，因此两次释放这块内存很可能造成灾难性的后果（见 12.4.6 节）。

13.3.1 拷贝构造函数

那么，我们应该怎么做呢？答案很明显：提供一个复制元素的拷贝操作，并确保当我们用一个 `vector` 初始化另一个 `vector` 时，这个拷贝操作会被调用。

一个类的对象的初始化是由该类的构造函数实现的。因此，我们需要一个进行拷贝操作的构造函数。不出意料，这种构造函数被称为拷贝构造函数（copy constructor）。它应接受待拷贝对象的引用作为参数。因此，对类 `vector`，它的拷贝构造函数为如下形式：

```
vector(const vector&);
```

我们试图使用一个 `vector` 初始化另一个 `vector` 时，这一拷贝构造函数就会被调用。拷贝构造函数使用对象引用作为参数的原因在于我们（显然）不希望在传递函数参数时又发生参数的拷贝，而使用 `const` 引用的原因在于我们不希望函数对参数进行修改（见 8.5.6 节）。因此，我们重新定义 `vector` 如下：

```

class vector {
    int sz;
    double* elem;
public:
    vector(const vector&);           // 拷贝构造函数：定义拷贝操作
    ...
};

```

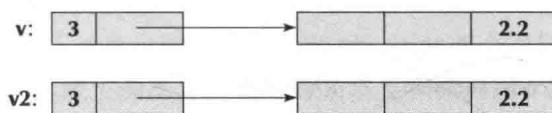
在从参数 `vector` 拷贝元素值之前，拷贝构造函数会设置元素数量 (`sz`)，为元素分配内存（初始化 `elem`）：

```
vector:: vector(const vector& arg)
// 分配元素，然后通过拷贝初始化它们
:sz{arg.sz}, elem{new double[arg.sz]}
{
    copy(arg.elem, arg.elem+sz, elem); // std::copy(), 参见附录 C.5.2
}
```

有了这个拷贝构造函数，我们再次考虑之前的例子：

```
vector v2 = v;
```

此定义会初始化 `v2`，这是通过调用 `vector` 的拷贝构造函数并将 `v` 作为参数传递给它而完成的。我们再以三个元素的 `vector` 为例，现在 `v` 和 `v2` 可图示如下：



这样，析构函数就能正确完成清理工作了，每个元素都会被正确释放。显然，现在两个 `vector` 是相互独立的，因此我们改变 `v` 的元素值而不会影响到 `v2`，反之亦然。例如：

```
v.set(1,99);           // 将 v[1] 设置为 99
v2.set(0,88);          // 将 v2[0] 设置为 88
cout << v.get(0) << ' ' << v2.get(1);
```

这段代码将输出 0 0。

除了如下进行拷贝构造：

```
vector v2 = v;
```

我们还可以使用下面这种等价形式

```
vector v2 {v};
```

当 `v`（初始化器）与 `v2`（被初始化变量）为相同类型且该类型定义了拷贝构造函数时，则这两种初始化方式完全相同，你可以选择自己更喜欢的方式。

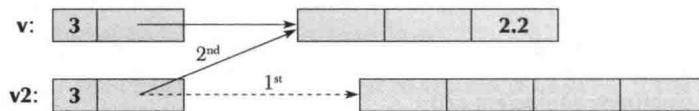
13.3.2 拷贝赋值

我们可以通过构造函数拷贝（初始化）对象，但我们也可以通过赋值的方式拷贝 `vector`。~~X~~ 与拷贝初始化类似，默认的拷贝赋值是逐成员的拷贝。因此，对于我们目前定义的 `vector`，拷贝赋值会造成双重释放（如 13.2.1 节中的拷贝构造函数所示）以及内存泄漏问题。例如：

```
void f2(int n)
{
    vector v(3);           // 定义一个 vector
    v.set(2,2.2);
    vector v2(4);
    v2 = v;               // 赋值：会发生什么？
    ...
}
```

我们希望 `v2` 成为 `v` 的副本（标准库 `vector` 就是这样做的），但由于我们并未给 `vector` 定义拷贝赋值操作，因此将执行默认的拷贝赋值操作；即赋值操作将进行逐成员拷贝，因此，`v2` 的

`sz`、`elem` 将会与 `v` 的 `sz`、`elem` 完全相同，如下所示：



当我们离开 `f2()` 时，将会发生灾难性错误，与 13.2 节中我们添加拷贝构造函数之前离开 `f()` 时一样：被 `v` 与 `v2` 共同指向的元素将被释放两次（使用 `delete[]`）。另外，还会发生内存泄漏：我们“忘记了”释放最初为 `v2` 的四个元素所分配的内存。对拷贝赋值操作的改进本质上与对拷贝初始化的改进（见 13.2.1 节）完全相同。我们应像下面代码这样恰当定义拷贝赋值操作：

```
class vector {
    int sz;
    double* elem;
public:
    vector& operator=(const vector&); // 拷贝赋值
    ...
};

vector& vector::operator=(const vector& a)
// 将本 vector 变为 a 的副本
{
    double* p = new double[a.sz]; // 分配新空间
    copy(a.elem,a.elem + a.sz,p); // 拷贝元素
    delete[] elem; // 释放旧空间
    elem = p; // 现在我们可以重置 elem 了
    sz = a.sz;
    return *this; // 返回一个自引用 (见 12.10 节)
}
```

赋值比构造稍微复杂一些，因为我们必须处理旧有元素。我们的基本策略是创建源 `vector` 元素的一份拷贝：

```
double* p = new double[a.sz]; // 分配新空间
copy(a.elem,a.elem + a.sz,p); // 拷贝元素
```

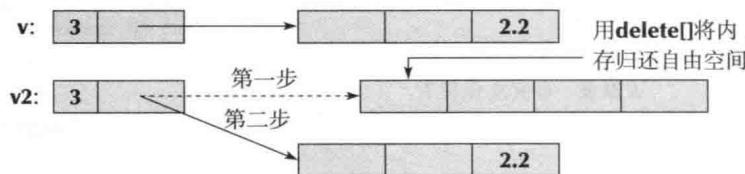
然后，我们将释放目标 `vector` 的旧有元素：

```
delete[] elem; // 释放旧空间
```

最后，我们将 `elem` 指向新元素：

```
elem = p; // 现在我们可以重置 elem 了
sz = a.sz;
```

结果可图示如下：



现在，我们的 `vector` 已不存在内存泄漏以及内存重复释放（`delete[]`）问题。

在实现拷贝赋值操作时，你可以在创建副本之前先释放旧有元素所占用的内存以简化代码，但通常更好的做法是在确信可以替换指定信息之前不要丢掉它。而且，如果你这么做了，在将一个 `vector` 赋值给它自身时就会产生奇怪的结果：

```
vector v(10);
v = v; // 自赋值
```

请仔细检查我们的实现，确保它能正确地处理这种情况（不考虑性能是否最优化）。

13.3.3 拷贝术语

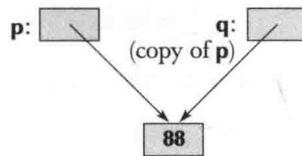
对于大多数程序员和大多数程序设计语言，拷贝都是一个重要问题。一个基本问题是你应该拷贝一个指针（或引用）还是应该拷贝指针指向（或引用）的数据：

- 浅拷贝（shallow copy）只拷贝指针，因此两个指针会指向同一个对象。指针和引用类型就是进行浅拷贝。
- 深拷贝（deep copy）将拷贝指针指向的数据，因此两个指针将指向两个不同的对象。`vector` 与 `string` 都实现了深拷贝。当类对象需要深拷贝时，我们需要为其定义拷贝构造函数和拷贝赋值操作。

下面是一个浅拷贝的例子：

```
int* p = new int(77);
int* q = p; // 拷贝指针 p
*p = 88; // 改变 p 和 q 指向的 int 的值
```

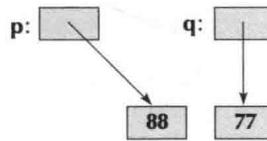
结果可图示如下：



与之相对，我们也可以进行深拷贝：

```
int* p = new int(77);
int* q = new int(*p); // 分配一个新的 int，然后拷贝 p 指向的值
*p = 88; // 改变 p 指向的 int 的值
```

结果如下图所示：



从拷贝术语可以看出，我们原来的 `vector` 的问题在于它只实现了浅拷贝，而不是拷贝指针 `elem` 指向的元素。而改进的 `vector` 则与标准库 `vector` 相似，实现了深拷贝：它为元素分配新的内存空间并进行元素的拷贝。实现了浅拷贝的类型（如指针与引用）被称为具有指针语义（pointer semantic）或引用语义（reference semantic）（它们拷贝地址）。实现了深拷贝的类型（如 `vector` 和 `string`）被称为具有值语义（value semantic）（它们拷贝指向的值）。从用户的角度看来，具有值语义的类型的行为就像没有涉及指针一样——只涉及能被拷贝的值。我

们可以从另一个角度思考具有值语义的类型：当谈到拷贝时，它们的行为“就像整数一样”。

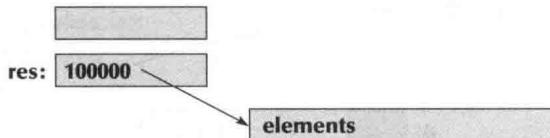
13.3.4 移动

如果一个 `vector` 有很多元素，那么拷贝的代价会很高。因此，我们只应在必要时才拷贝 `vector`。考虑下面这个例子：

```
vector fill(istream& is)
{
    vector res;
    for (double x; is>>x; ) res.push_back(x);
    return res;
}

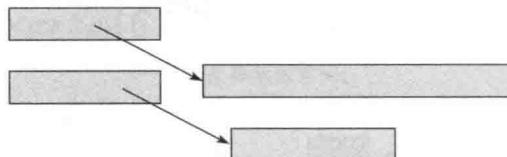
void use()
{
    vector vec = fill(cin);
    // ... 使用 vec ...
}
```

在本例中，我们从输入流读取数据存入 `res`，然后将它返回给 `use()`。将 `res` 从 `fill()` 拷贝出来并拷贝到 `vec` 中，代价可能很高。但为什么要拷贝呢？我们不需要拷贝！在从函数返回后，我们不可能再使用原对象了（`res`）。实际上，`res` 会被销毁，这是 `fill()` 返回过程的一部分。那么我们如何避免拷贝呢？让我们再次考察一个向量在内存中的表示：



我们希望能“偷出”`res` 的表示用于 `vec`。换句话说，我们希望 `vec` 指向 `res` 的元素，而不进行任何拷贝。

在将 `res` 的元素指针和元素数量移动到 `vec` 后，`vec` 就持有了元素，我们就成功地完成了将 `res` 中的元素值移出 `fill()` 并移到 `vec` 中的工作。现在，`res` 可以被（简单且高效地）销毁，没有任何不良副作用：



我们成功地将 100 000 个 `double` 移出 `fill()` 并移到它的调用者中，而代价仅仅是四个机器字的赋值。

我们如何用 C++ 代码表达这种移动？我们可以定义移动操作，作为拷贝操作的补充：

```
class vector {
    int sz;
    double* elem;
public:
    vector(vector&& a);           // 移动构造函数
    vector& operator=(vector&&); // 移动赋值
    ...
};
```

滑稽的 `&&` 符号被称为“右值引用”。我们用它来定义移动操作。注意，移动操作不接受 `const` 参数；即，我们应使用 `(vector&&)` 而不是 `(const vector&&)`。移动操作的目的之一是修改源对象，令其变为“空的”。移动操作的定义应该更简单，应该比对应的拷贝操作更简单高效。对于 `vector`，我们有

```
vector::vector(vector&& a)
    :sz{a.sz}, elem{a.elem}           // 拷贝 a 的 elem 和 sz
{
    a.sz = 0;                      // 令 a 变为空 vector
    a.elem = nullptr;
}

vector& vector::operator=(vector&& a) // 将 a 移动到本 vector
{
    delete[] elem;                // 释放旧空间
    elem = a.elem;                // 拷贝 a 的 elem 和 sz
    sz = a.sz;                    // 令 a 变为空 vector
    a.elem = nullptr;
    a.sz = 0;
    return *this;                 // 返回一个自引用（见 12.10 节）
}
```

通过定义一个移动构造函数，我们令移动大量信息（例如移动包含很多元素的向量）变得更容易更高效。再次考虑 `fill()` 函数：

```
vector fill(istream& is)
{
    vector res;
    for (double x; is>>x; ) res.push_back(x);
    return res;
}
```

移动构造函数隐式地用于实现函数返回。编译器知道要返回的局部值 (`res`) 将要离开其作用域，因此可以将其值移出而非拷贝它。

移动构造函数的重要性在于我们不必为了从一个函数返回大量信息而处理指针或引用。 考虑下面这种有瑕疵的（但很常见的）替代方法：

```
vector* fill2(istream& is)
{
    vector* res = new vector;
    for (double x; is>>x; ) res->push_back(x);
    return res;
}

void use2()
{
    vector* vec = fill(cin);
    // ... 使用 vec ...
    delete vec;
}
```

若采用这种方法，我们就必须记得释放 `vector`。如 12.4.6 节所述，释放自由空间上的对象并不像看起来那么容易实现一致性和正确性。

13.4 必要的操作

在学习了前几节的知识之后，我们现在可以讨论如何决定一个类应定义哪些构造函 

数、是否应定义析构函数、是否应定义拷贝和移动操作这些问题了。有七种必要的操作需要考虑：

- 接受一个或多个参数的构造函数。
- 默认构造函数。
- 拷贝构造函数（拷贝同一类型的对象）。
- 拷贝赋值操作（拷贝同一类型的对象）。
- 移动构造函数（移动同一类型的对象）。
- 移动赋值操作（移动同一类型的对象）。
- 析构函数。

通常，我们需要一个或多个构造函数，接受不同参数来初始化对象。例如：

```
string s {"cat.jpg"};           // 将 s 初始化为字符串 "cat.jpg"
Image ii {Point{200,300}, "cat.jpg"}; // 用坐标 {200, 300} 初始化一个 Point
                                         // 然后在这个位置显示文件 cat.jpg 的内容
```

初始化器的含义 / 用途完全取决于构造函数。标准 `string` 的构造函数使用一个字符串作为初始值，而 `Image` 的构造函数使用此字符串作为要打开的文件名。通常我们使用构造函数来建立不变式（见 9.4.3 节）。如果我们无法为类定义一个好的不变式，使得构造函数能建立它的话，那么很可能这个类的设计很糟糕或者它只是一个普通数据结构。

接受参数的构造函数随着所属类的不同而不同，其他操作的形式则更加规则。

我们如何知道一个类是否需要默认构造函数呢？如果我们希望在不指定初始化器的前提下还能构造一个类的对象，那么该类就需要默认构造函数。最常见的例子是我们希望将某个类的对象存放在标准库的 `vector` 对象之中。下面的这些代码是正确的，因为 `int`、`string` 和 `vector<int>` 都具有默认值：

```
vector<double> vd(10);          // 10 个 double 的 vector，每个 double 都初始化为 0.0
vector<string> vs(10);          // 10 个 string 的 vector，每个 string 都初始化为 ""
vector<vector<int>> vvi(10);    // 10 个 vector 的 vector，每个 vector 都初始化为 vector()
```

因此，默认构造函数常常是有用的。问题现在变为：“何时拥有一个默认构造函数是有意义的？”一个答案是：“当我们可以通过一个有意义的、显然的默认值来为类建立起不变式时。”对于值类型，如 `int` 和 `double`，显然的默认值为 0（对于 `double`，为 0.0）。对于 `string`，默认值为空字符串 `""`，这也是显然的。对于 `vector`，默认值为空向量。对于每个类型 `T`，若存在默认值，则 `T{} 为默认值`。例如，`double{}` 为 0.0，`string{}` 为 `""`，`vector<int>()` 为一个空的 `int` 的 `vector`。

如果一个类需要获取资源，则它需要析构函数。所谓资源，就是一种你“从某处获取”，且使用完毕后必须归还的东西。一个明显的例子是你（用 `new`）从自由空间获取的内存，用完后必须归还自由空间（用 `delete` 或者 `delete[]`）。我们的 `vector` 需要获取内存资源以保存它的元素，因此它必须在使用完毕后归还内存，这样它就需要一个析构函数。随着程序规模和复杂性的增长，你可能遇到的其他资源包括文件（如果你打开了一个文件，就需要负责将它关闭）、锁、线程句柄以及套接字（用于远端计算机或进程间的通信）。

一个类需要析构函数的另一个简单标志是它包含指针成员或引用成员。如果一个类具有指针成员或引用成员，则它通常需要一个析构函数以及拷贝操作。

一个需要析构函数的类几乎肯定也需要一个拷贝构造函数和一个拷贝赋值操作。其原因很简单，如果一个对象获取了一种资源（并有一个指向资源的指针成员），那么只进行默认

拷贝（浅拷贝，逐成员拷贝）几乎肯定会带来错误。`vector` 就是一个典型的例子。

类似地，一个需要析构函数的类几乎肯定也需要一个移动构造函数和一个移动赋值操作。其原因很简单，如果一个对象获取了一种资源（并有一个指向资源的指针成员），那么只进行默认拷贝（浅拷贝，逐成员拷贝）几乎肯定会带来错误，而常用的补救方法（复制完整对象状态的拷贝操作）可能代价很高。`vector` 就是一个典型的例子。

另外，对于一个基类而言，如果它的派生类具有析构函数，则该基类需要一个 `virtual` 析构函数（见 12.5.2 节）。

13.4.1 显式构造函数

只接受一个参数的构造函数定义了一个从其参数类型向所属类的类型转换操作。这种转换是很有用的。例如：

```
class complex {
public:
    complex(double);           // 定义了 double 向 complex 的类型转换
    complex(double,double);
    ...
};

complex z1 = 3.14;           // 正确：将 3.14 转换为 (3.14, 0)
complex z2 = complex{1.2, 3.4};
```

但是，我们应谨慎地使用隐式转换，因为隐式转换可能会造成不可预料的后果。例如，我们目前定义的 `vector` 有一个接受 `int` 参数的构造函数。这意味着它定义了一个从 `int` 向 `vector` 的类型转换操作。例如：

```
class vector {
    ...
    vector(int);
    ...
};

vector v = 10;               // 奇怪：创建了一个 10 个 double 的 vector
v = 20;                     // 啊？将一个包含 20 个 double 的新 vector 赋予 v

void f(const vector&);
f(10);                      // 啊？用一个包含 10 个 double 的新 vector 调用 f
```

看起来好像我们获得了预料之外的东西。幸运的是，我们能够通过一种简单的方式禁止将构造函数用于隐式类型转换。由关键字 `explicit` 定义的构造函数（即显式构造函数）只能用于对象的构造而不能用于隐式转换。例如：

```
class vector {
    ...
    explicit vector(int);
    ...

vector v = 10;               // 错误：不存在 int 到 vector 的转换
v = 20;                     // 错误：不存在 int 到 vector 的转换
vector v0(10);              // 正确

void f(const vector&);
f(10);                      // 错误：不存在 int 到 vector<double> 的转换
f(vector(10));              // 正确
```

为了避免意外的类型转换，我们和标准库都将 `vector` 的单参数构造函数定义为 `explicit` 的。很遗憾，构造函数默认不是 `explicit` 的；当我们拿不定主意时，应将所有单参数的构造函数定义为 `explicit` 的。

13.4.2 调试构造函数和析构函数

在程序的执行过程中，构造函数与析构函数都将在明确的、可预计的时间点上被调用。但是，我们并不总是以显式方式（如 `vector(2)`）调用它们，我们在做某些事的时候也会调用这些函数——如声明一个 `vector` 对象、以传值方式传递一个 `vector` 参数或者用 `new` 在自由空间上创建一个 `vector`。这可能会造成人们语法上的混淆，因为不只有一种触发构造函数的语法。我们可以更简单地看待构造函数和析构函数的调用：

- 每当类型 `X` 的一个对象被构建时，类型 `X` 的一个构造函数将被调用。
- 每当类型 `X` 的一个对象被销毁时，类型 `X` 的析构函数将被调用。

每当一个类对象被销毁时，该类的析构函数将被调用；这种情况可能发生在变量的作用域结束时、程序结束时或者 `delete` 用于一个指向对象的指针时。每当一个类对象被构建时，该类的一个（恰当的）构造函数将被调用；这种情况可能发生在变量初始化时、用 `new` 创建对象（内置类型除外）时，以及拷贝对象时。

但这些情况什么时候会发生？体会它们的一个好办法是向构造函数、赋值操作和析构函数添加打印语句，然后尝试运行程序。例如：

```
struct X {           // 简单的测试类
    int val;

    void out(const string& s, int nv)
    { cerr << this << " ->" << s << ":" << val << "(" << nv << ")\\n"; }

    X() { out("X()",0); val=0; }           // 默认构造函数
    X(int v) { val=v; out("X(int)",v); }
    X(const X& x){ val=x.val; out("X(X&)",x.val); }   // 拷贝构造函数
    X& operator=(const X& a)               // 拷贝赋值操作
        { out("X::operator=(())",a.val); val=a.val; return *this; }
    ~X() { out("~X()",0); }                 // 析构函数
};
```

这样，我们对 `X` 做的任何事情都会留下踪迹以供学习。例如：

```
X glob(2);           // 一个全局变量

X copy(X a) { return a; }

X copy2(X a) { X aa = a; return aa; }

X& ref_to(X& a) { return a; }

X* make(int i) { X a(i); return new X(a); }

struct XX { X a; X b; };

int main()
{
    X loc {4};           // 局部变量
    X loc2 {loc}; // 拷贝构造函数
```

```

loc = X{5};           // 拷贝赋值操作
loc2 = copy(loc);    // 传值调用并返回
loc2 = copy2(loc);
X loc3 {6};
X& r = ref_to(loc); // 传引用调用并返回
delete make(7);
delete make(8);
vector<X> v(4);    // 默认值
XX loc4;
X* p = new X(9);    // 在自由空间上分配一个 X
delete p;
X* pp = new X[5];   // 在自由空间上分配一个 X 的数组
delete[] pp;
}

```

试一试执行这一程序。

试一试

运行这一程序示例并确保你能够弄清结果的含义。如果你这么做了，你就能够明白对象的构造与析构的大致过程。

依赖于你所使用的编译器的质量，你可能会发现一些与调用 `copy()` 和 `copy2()` 有关的“误拷贝”。我们（人类）可以看出这两个函数并没有做任何有意义的事情：它们仅仅将值原封不动地从函数输入拷贝到函数输出。如果编译器足够聪明，能发现这一事实，那么它可以去掉对拷贝构造函数的调用。换句话说，编译器有权认为一个拷贝构造函数除了拷贝什么也不做。一些足够聪明的编译器能消除很多无谓的拷贝。但是，编译器并不保证有这么聪明，因此如果你希望程序的性能跨平台也不变，可以考虑移动操作（见 13.3.4 节）。

现在思考一个问题：我们为什么要为这样一个“愚蠢的类 X”费心呢？这有点像音乐家所必须做的指法练习。在做了这些简单的事情之后，其他事情——真正有意义的事情也就变得容易了。而且，如果你对构造函数和析构函数存有疑问，那么你也可以在自己实际的类的构造函数中插入这种打印语句来观察它们是否如预期那样奏效。对于规模更大的程序，这种跟踪方法有些繁琐恼人，但可使用类似的技术。例如，你可以通过观察构造函数的调用次数与析构函数的调用次数是否相等来判断程序中是否存在内存泄漏问题。对于分配资源或包含指针成员的类，忘记定义拷贝构造函数和拷贝赋值操作是常见的问题根源，但这是很容易避免的。

如果你的程序的规模变得太大，难以用这种简单的方法进行处理，那么你就需要学会使用一些专业的工具来发现这类问题；这些工具通常被称为“泄漏探测器”。当然，最理想的情况是使用一些避免内存泄漏的技术来防止泄漏发生。

13.5 访问 vector 元素

到目前为止（12.6 节），我们已经使用过成员函数 `set()` 和 `get()` 访问 `vector` 的元素，但这种用法冗长、不美观。我们希望能够使用习惯的下标表示方式 `v[i]`。为此，我们需要定义一个名为 `operator[]` 的成员函数。下面是我们的初次（简单）尝试：

```

class vector {
    int sz;           // 大小

```

```

double* elem;           // 指向元素的指针
public:
    ...
    double operator[](int n) { return elem[n]; } // 返回元素
};

```

这个定义看起来令人满意而且很简单，但不幸的是它过于简单了。下标操作(`operator[]()`)返回一个值，因此只能实现对元素的读操作而未实现写操作：

```

vector v(10);
double x = v[2];           // 很好
v[3] = x;                // 错误：v[3] 不是一个左值

```

在这段代码中，`v[i]`被解释为函数调用`v.operator[](i)`，返回`v`的编号为`i`的元素的值。对于过于简单的`vector`，`v[3]`是一个浮点类型的数值，而不是一个浮点类型的变量。

试一试

编写此版本`vector`的完整实现，使之能进行编译，并观察编译器对语句“`v[3]=x;`”会报告怎样的错误消息。

我们进一步尝试令`operator[]`返回指向对应元素的指针：

```

class vector {
    int sz;                  // 大小
    double* elem;            // 指向元素的指针
public:
    ...
    double* operator[](int n) { return &elem[n]; } // 返回指针
};

```

有了这个定义，我们可以编写如下代码：

```

vector v(10);
for (int i=0; i<v.size(); ++i) { // 正确，但太不美观
    *v[i] = i;
    cout << *v[i];
}

```

`v[i]`在这里被解释为函数调用`v.operator[](i)`，返回指向`v`的编号为`i`的元素的指针。这种实现的问题是，在我们对元素进行访问时，不得不用`*`对指针进行解引用。这与必须使用`set()`和`get()`几乎一样糟糕。从下标运算符返回元素的引用可以解决这一问题：

```

class vector {
    ...
    double& operator[](int n) { return elem[n]; } // 返回引用
};

```

现在，我们可以编写如下代码：

```

vector v(10);
for (int i=0; i<v.size(); ++i) { // 正确！
    v[i] = i; // v[i] 返回元素 i 的引用
    cout << v[i];
}

```

我们已经实现了传统表示方法：`v[i]`被解释为函数调用`v.operator[](i)`，返回`v`的编号为`i`的元素的引用。

13.5.1 对 **const** 向量重载运算符

到目前为止，`operator[]()` 的定义存在一个问题：它不能用于 `const vector` 对象。例如：

```
void f(const vector& cv)
{
    double d = cv[1];           // 错误，但本应是正确的
    cv[1] = 2.0;                // 错误（本该如此）
}
```

其原因在于我们的 `vector::operator[]()` 可能会潜在地改变 `vector` 对象。即使它实际上没有改变，编译器仍会认为这是一个错误，因为我们“忘了”将这一情况告诉编译器。解决方法是再定义一个 `const` 成员函数（参见 9.7.4 节）的版本。这很容易实现：

```
class vector {
    ...
    double& operator[](int n);      // 用于非 const 的 vector
    double operator[](int n) const; // 用于 const vector
};
```

对 `const` 版本，我们显然不能返回一个 `double&`，而应返回一个 `double` 值。返回一个 `const double&` 的效果是一样的，但由于 `double` 只是一个很小的对象，没有必要返回引用（见 8.5.6 节），因此我们决定以传值方式返回它。现在，我们可以编写如下代码：

```
void ff(const vector& cv, vector& v)
{
    double d = cv[1];           // 正确（使用 const []）
    cv[1] = 2.0;                // 错误（使用 const []）
    double d = v[1];           // 正确（使用非 const []）
    v[1] = 2.0;                // 正确（使用非 const []）
}
```

由于 `vector` 对象常常通过 `const` 引用的方式传递，因此为 `operator[]()` 实现 `const` 版本是十分必要的。

13.6 数组

我们用数组（array）来表示自由空间中分配的对象序列已经有一段时间了。与具名变量一样，我们也可以在其他的地方分配数组。实际上，数组通常可定义为

- 全局变量（但定义全局变量通常是一个糟糕的主意）；
- 局部变量（但数组作为局部变量有严重的局限）；
- 函数参数（但一个数组不知道其自身大小）；
- 类的成员（但数组成员难于初始化）。

现在，你可能会发觉我们明显更倾向于使用 `vector` 而不是数组。如果可以选择，你应当尽可能地使用 `Std::vector`，而在大多数场景下你是有选择权的。但是，数组在 `vector` 出现之前就已经存在很长时间了，并且它与其他编程语言（尤其是 C 语言）中提供的数组大致相同，因此你必须了解数组，而且必须深入了解，以便能处理旧代码和未意识到 `vector` 优点的人编写的代码。

那么，什么是数组呢？我们该如何定义数组？又该如何使用数组？一个数组就是内存中连续存储的同构对象序列；也就是说，一个数组中的所有元素都具有相同的类型，并且各元素之间不存在内存间隙。数组中的元素从 0 开始顺序编号。在声明中，我们用“方括号”表

示数组：

```
const int max = 100;
int gai[max];           // 一个全局数组（包含 100 个 int）；“永远活跃”

void f(int n)
{
    char lac[20];      // 局部数组；“活跃”至作用域结束为止
    int lai[60];
    double lad[n];    // 错误：数组大小不是常量
    // ...
}
```

注意数组使用的限制：一个具名数组的元素数目必须在编译时就已知。如果你希望元素的数目是一个变量，那么就必须在自由空间中分配数组，并通过指针对数组进行访问。这正是 `vector` 对其元素数组所做的。

像在自由空间中的数组一样，我们通过下标与解引用运算符（`[]` 和 `*`）访问具名数组。例如：

```
void f2()
{
    char lac[20];      // 局部数组；“活跃”至作用域结束为止

    lac[7] = 'a';
    *lac = 'b';        // 等价于 lac[0]='b'

    lac[-2] = 'b';     // 啊？
    lac[200] = 'c';    // 啊？
}
```

⚠ 此函数能够通过编译，但我们知道，“通过编译”并不意味着函数能够“正确工作”。运算符 `[]` 的使用是显而易见的，但范围检查却没有进行。因此，虽然函数 `f2()` 通过了编译，但对 `lac[-2]` 和 `lac[200]` 进行写操作的后果是灾难性的，我们应避免这样的操作。数组不会进行范围检查。再次强调，在本例中我们在直接处理物理内存，不要期待“系统支持”。

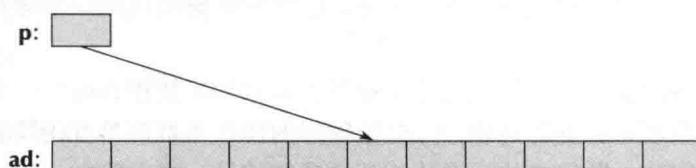
⚠ 难道编译器就不能发现 `lac` 只有 20 个元素从而判定 `lac[200]` 是错误的吗？编译器能够做到这一点，但据我们所知，到目前为止还没有哪个编译器产品实现了这样的功能。问题在于在编译时跟踪数组的范围一般而言是不可能的，而只查找最简单情况下的错误（如上面的错误）并不是十分有用。

13.6.1 指向数组元素的指针

指针可以指向数组的元素。例如：

```
double ad[10];
double* p = &ad[5];      // 指向 ad[5]
```

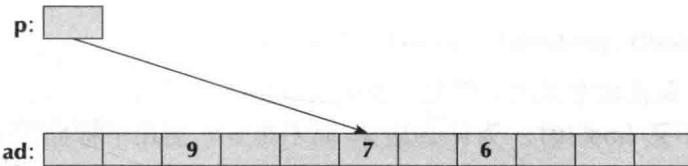
现在指针 `p` 指向 `double` 型元素 `ad[5]`：



我们能够对指针使用下标与解引用运算符：

```
*p = 7;  
p[2] = 6;  
p[-3] = 9;
```

我们有

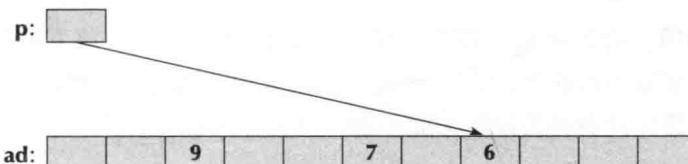


即，我们既可以用正数也可以用负数作为下标。只要结果元素位于数组的范围之内，就是合法的操作。但是，通过指针访问位于数组范围之外的数据是非法的（与自由空间上分配的数组一样，参见 12.4.3 节）。通常，编译器不能检测数组范围之外的访问，并且这样的访问（迟早）会造成灾难性的后果。

当指针指向一个数组内时，我们对它进行加法和下标操作就能令它指向数组中的其他元素。例如：

```
p += 2; // 将 p 向右移动 2 个元素
```

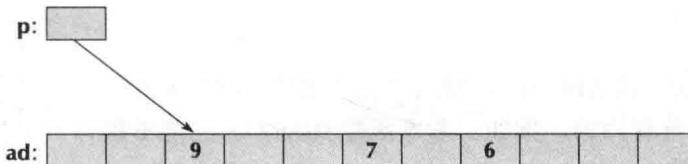
我们得到



下面语句

```
p -= 5; // 将 p 向左移动 5 个元素
```

会得到



用 +、-、+= 与 -= 移动指针称为指针运算 (pointer arithmetic)。显然，当我们进行这种运算时必须十分小心，确保结果不超出数组的范围：

```
p += 1000; // 疯狂：p 指向的数组只有 10 个元素  
double d = *p; // 非法：可能得到一个糟糕的值  
                // (肯定是一个不可预知的值)  
*p = 12.34; // 非法：可能破坏一些未知的数据
```

不幸的是，由指针运算所造成的错误有时很难被发现。通常最好的策略是尽量避免使用指针运算。

最常见的指针运算是对指针进行自增操作（使用 `++`）以使指针指向下一个元素，以及对指针进行自减操作（使用 `--`）以使指针指向下一个元素。例如，我们可以通过如下方式打印 `ad` 的元素的值：

```
for (double* p = &ad[0]; p < &ad[10]; ++p) cout << *p << '\n';
```

或者反向打印：

```
for (double* p = &ad[9]; p >= &ad[0]; --p) cout << *p << '\n';
```

这种指针运算是比较常见的。但是，我们发现后一个（“反向”）例子很容易出错。为什么是 `&ad[9]` 而不是 `&ad[10]`？为什么是 `>=` 而不是 `>`？使用下标操作也能很好地、很高效地实现这些例子。使用 `vector` 和下标操作也能很好地实现这些例子，而且更容易实现范围检查。

注意，指针运算在实际程序中最常见的用途是对传递给函数的指针参数进行运算。在这种情况下，编译器并不知道指针指向的数组包含元素的个数：一切都要靠你自己把握。只要我们能够选择，最好能避免这种情况。

为什么 C++ 允许进行指针运算呢？指针运算可能造成这么大的麻烦，而且与下标操作相比不能提供任何更多的功能。例如：

```
double* p1 = &ad[0];
double* p2 = p1 + 7;
double* p3 = &p1[7];
if (p2 != p3) cout << "impossible!\n";
```

 主要是历史原因。指针运算在很久以前就在 C 语言中存在，将其剔除会造成很多代码不能够运行。还有部分原因在于，在一些重要低层应用（如内存管理器）中，使用指针运算更为便利。

13.6.2 指针和数组

 数组的名字代表了数组的所有元素。例如：

```
char ch[100];
```

`ch` 的大小 (`sizeof(ch)`) 为 100。然而，数组的名字可以转化（退化）为指针。例如：

```
char* p = ch;
```

`p` 被初始化为 `&ch[0]`，而 `sizeof(p)` 可能为 4 之类的值（而非 100）。

这一特性是十分有用的。例如，考虑函数 `strlen()`，它能够统计一个以 0 结尾的字符数组中包含的字符总数：

```
int strlen(const char* p) // 类似标准库 strlen()
{
    int count = 0;
    while (*p) { ++count; ++p; }
    return count;
}
```

我们可以调用 `strlen(ch)`，也可以调用 `strlen(&ch[0])`。你可能认为这只不过是符号表示上的一个小小优点，对此我必须表示同意。

将数组名转化为指针的一个原因是避免意外地以传值方式传递大量数据。例如：

```

int strlen(const char a[]) // 类似标准库 strlen()
{
    int count = 0;
    while (a[count]) { ++count; }
    return count;
}

char lots[100000];

void f()
{
    int nchar = strlen(lots);
    ...
}

```

你可能会天真地（也是非常合理地）认为这个 `strlen()` 调用会拷贝参数所指向的 100 000 个字符，但这并不会发生。取而代之的是，编译器会认为参数声明 `char p[]` 等价于 `char *p`，而函数调用 `strlen(lots)` 等价于 `strlen(&lots[0])`。这能帮你避免代价高昂的拷贝操作，但可能会令你感到惊讶。为什么呢？因为在其他的所有情况下，当你向函数传递一个对象，又未显示声明以引用方式（见 8.5.3 ~ 8.5.6 节）传递它时，对象都会被拷贝。

数组名会被当作指向其第一个元素的指针处理，注意，你通过这种方式获得的指针不是一个变量而是一个值，因此你不能对它进行赋值：

```

char ac[10];
ac = new char [20];      // 错误：不能为数组名赋值
&ac[0] = new char [20]; // 错误：不能为指针值赋值

```

最终，编译器将会发现这一错误！

作为数组名向指针隐式转换的一个结果，你不能通过赋值操作拷贝数组：

```

int x[100];
int y[100];
...
x = y;                  // 错误
int z[100] = y;          // 错误

```

这些特性是一致的，但对程序员通常是一个麻烦。如果你需要拷贝一个数组，就必须编写一些更复杂的代码来实现。例如：

```

for (int i=0; i<100; ++i) x[i]=y[i]; // 拷贝 100 个 int
memcpy(x,y,100*sizeof(int)); // 拷贝 100*sizeof(int) 个字节
copy(y,y+100, x);          // 拷贝 100 个 int

```

注意，C 语言不支持像 `vector` 这样的类型，因此在 C 中，你必须广泛使用数组。这意味着仍有很多的 C++ 代码使用数组（见 27.1.2 节）。特别地，C 风格字符串（以 0 结尾的字符数组，参见 27.5 节）十分常见。

如果你希望进行数组赋值，就必须使用像 `vector` 这样的类型。等价于上述拷贝代码的 `vector` 实现为：

```

vector<int> x(100);
vector<int> y(100);
...
x = y;                  // 拷贝 100 个 int

```

13.6.3 数组初始化



`char` 数组可用字符串字面值常量初始化。例如：

```
char ac[] = "Beorn"; // 6 个字符的数组
```

数一数字符数，只有 5 个，但 `ac` 是一个含有 6 个字符的数组，因为编译器会在字符串字面值常量的末尾添加一个字符 0 表示结束：

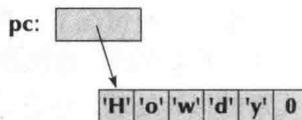
ac:

'B'	'e'	'o'	'r'	'n'	0
-----	-----	-----	-----	-----	---

字符串以 0 结尾在 C 语言和很多系统中是规范表示方法。我们称这种以 0 结尾的字符数组为 C 风格字符串（C-style string）。所有字符串字面值常量都是 C 风格字符串。例如：

```
char* pc = "Howdy"; // pc 指向一个 6 个字符的数组
```

该字符串可图示如下：



注意，数值为 0 的 `char` 不是字符 ‘0’ 或者其他的任何字母或数字。这种结尾 0 的目的在于帮助函数定位字符串的结束。记住，数组并不知道自身大小。依赖于 0 结尾这一规范，我们可以编写如下代码：

```
int strlen(const char* p) // 类似标准库 strlen()
{
    int n = 0;
    while (p[n]) ++n;
    return n;
}
```

实际上，我们不需要自己实现 `strlen()`，因为它是一个标准库函数，定义在头文件 `<string.h>` 中（见 27.5 节和附录 C10.3）。注意，`strlen()` 只统计字符数，并不统计结尾的 0；也就是说，我们需要 $n+1$ 个 `char` 以存储 n 个字符的 C 风格字符串。

只有字符数组能用字符串字面值常量进行初始化，但所有数组都能用一个其元素类型值的列表进行初始化。例如：

```
int ai[] = { 1, 2, 3, 4, 5, 6 }; // 6 个 int 的数组
int ai2[100] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }; // 后 90 个元素被初始化为 0
double ad[100] = { }; // 所有元素被初始化为 0.0
char chars[] = { 'a', 'b', 'c' }; // 无结尾 0 !
```

注意，`ai` 的元素个数为 6（不是 7）且 `chars` 的元素个数为 3（而非 4）——“在末尾加 0”这一规则只适用于字符串字面值常量。如果在初始化时未明确指定数组大小，那么编译器将通过初始化器列表推断其大小。这是相当有用特性。如果初始化器值的个数小于数组的实际大小（如 `ai2` 和 `ad` 的定义），剩下元素将被设置为该元素类型的默认值。

13.6.4 指针问题

与数组类似，指针经常被过度使用以及误用。这类问题通常同时涉及指针和数组，我们

将在本节总结这些问题。特别地，指针相关的所有严重问题通常都涉及意外访问：试图访问的东西不是期待类型的对象，并且其中很多问题都涉及对数组范围之外数据的访问。在本节中我们主要考虑以下问题：

- 使用空指针进行数据访问。
- 使用未初始化的指针进行数据访问。
- 对数组结尾之后的数据进行访问。
- 对已释放对象的访问。
- 对已离开作用域的对象进行访问。

在所有的情况下，对于编程者而言，一个实际的问题是所有真正的访问语句看起来都完全无辜；问题“不过”是指针被赋予的值令指针的使用变成了非法。更糟糕的是（通过指针写数据的情况下），这些问题可能会在很长时间后才会显现，而表现出来的现象是一些明显无关的对象被破坏。让我们考虑下面这个例子：

不要用空指针进行数据访问：

```
int* p = nullptr;
*p = 7;      // 糟糕！
```

显然，在实际程序中，这种问题通常发生在指针初始化和指针使用之间还有一些其他代码的情况下。特别地，向一个函数传递 `p` 然后又从函数接收 `p` 作为返回结果是很常见的。我们建议不要向函数传递空指针，但如果你不得不这么做，应在使用之前检测空指针：

```
int* p = fct_that_can_return_a_nullptr();

if (p == nullptr) {
    // 执行一些操作
}
else {
    // 使用 p
    *p = 7;
}
```

并且

```
void fct_that_can_receive_a_nullptr(int* p)
{
    if (p == nullptr) {
        // 执行一些操作
    }
    else {
        // 使用 p
        *p = 7;
    }
}
```

使用引用代替指针（见 12.9.1 节）和使用异常来报告错误（见 5.6 节和 14.5 节）是避免空指针的主要工具。

对你的指针进行初始化：

```
int* p;
*p = 9;      // 糟糕！
```

特别地，不要忘了对作为类成员的指针进行初始化。

不要访问不存在的数组元素：

```
int a[10];
int* p = &a[10];
*p = 11;      // 糟糕!
a[10] = 12;    // 糟糕!
```

在一个循环中访问第一个元素以及最后一个元素时应特别小心，应尽量避免将数组当作指向其第一个元素的指针进行传递。我们可以用 `vector` 代替数组。如果你确实需要在多个函数中使用一个数组（将它作为参数传递），那么就应极其小心并同时传递其大小。

 不要通过一个已清除的指针访问数据：

```
int* p = new int{7};
// ...
delete p;
// ...
*p = 13;      // 糟糕!
```

代码 `delete p` 或之后的代码可能胡乱修改了 `*p` 或将它用于了其他对象。在所有问题中，我们认为这一问题是难以系统地避免的。防止这一问题最有效的方法是不要使用“裸” `new` 操作，从而就不必使用“裸” `delete` 操作：只在构造函数和析构函数中使用 `new` 与 `delete`，或者使用容器如 `Vector_ref`（见附录 E.4）来处理 `delete`。

 不要返回指向局部变量的指针：

```
int* f()
{
    int x = 7;
    // ...
    return &x;
}

// ...

int* p = f();
// ...
*p = 15;      // 糟糕!
```

从 `f()` 返回的语句或之后的代码可能胡乱修改了 `*p` 或将它用于了其他对象。造成这一问题的原因在于，一个函数的局部变量在进入函数时分配内存空间（在栈中），在函数退出时被释放。特别是，如果局部变量的类具有析构函数，则析构函数会被调用（见 12.5.1 节）。编译器有能力捕获与返回局部变量指针相关的大部分问题，但只有少部分编译器会这么做。

考虑一个逻辑上等价的例子：

```
vector& ff()
{
    vector x(7);    // 7 个元素
    // ...
    return x;
} // vector x 在此销毁

// ...

vector& p = ff();
// ...
p[4] = 15;      // 糟糕!
```

很少有编译器会捕获这种返回问题。

程序员常常会低估这些问题。然而，很多有经验的程序员都曾被这些简单的数组和指针问题的无数变形和组合所打败。解决的方法是不要在代码中随意使用指针、数组、new 和 delete。如果你在实际规模的程序中随意使用这些特性，光靠小心谨慎是不够的。取而代之的是，我们应使用 `vector`、RAII（Resource Acquisition Is Initialization，参见 14.5 节）以及其他的一些系统化方法来管理内存与其他资源。

13.7 实例：回文

我们已经展示了足够多的技术示例！让我们尝试一个小小的难题。回文（palindrome）是一种单词，它顺序拼写和逆序拼写的结果是相同的。例如，anna、petep 和 malayalam 都是回文，而 ida 和 homesick 不是回文。有两种方法判断一个单词是否是回文：

- 获得单词逆序拼写的副本，并将其与原单词进行比较。
- 判断单词的首字符与尾字符是否相同，然后判断第二个字符与倒数第二个字符是否相同，继续比较下去直到到达单词的中央。

在这一节中，我们将采取第二种方法。根据单词表示方式的不同以及跟踪字符比较进度方式的不同，我们可以通过多种方式实现这一思路。我们将编写一个检测单词是否是回文的小程序，它将采用不同的实现方法，以观察不同的语言特性是如何影响代码的形式和工作方式的。

13.7.1 使用 `string` 实现回文

首先，我们使用标准库 `string` 类型配合 `int` 类型的索引跟踪字符比较的进度：

```
bool is_palindrome(const string& s)
{
    int first = 0;           // 首字符索引
    int last = s.length() - 1; // 尾字符索引
    while (first < last) {   // 我们还未到达中央
        if (s[first] != s[last]) return false;
        ++first;             // 向前移动
        --last;               // 向后移动
    }
    return true;
}
```

当比较到达单词的中央且未发现不同的字符时，函数返回 `true`。我们建议，在你编写这段代码时，应保证代码在下列情况下都是正确的：当字符串不包含任何字符时，当字符串只包含一个字符时，以及当字符串包含奇数个或偶数个字符时。当然，我们不应只依靠逻辑分析来判断代码是否正确，而应该进行测试。我们可以按照下面的方式测试 `is_palindrome()`：

```
int main()
{
    for (string s; cin >> s;) {
        cout << s << " is";
        if (!is_palindrome(s)) cout << " not";
        cout << " a palindrome\n";
    }
}
```

基本上，我们使用 `string` 类型的原因在于“`string` 善于处理单词”。将空白符分隔的单

词读入字符串是很简单的，而且一个 `string` 清楚地知道自身的大小。如果我们希望用包含空白符的字符串测试 `is_palindrome()`，可以使用 `getline()`（见 11.5 节）读取字符串。这时，`ad ha` 和 `as df fd sa` 会被认为是回文。

13.7.2 使用数组实现回文

当我们不能使用 `string`（或 `vector`），不得不使用数组存储字符时，该如何实现回文检测呢？让我们看看下面的程序：

```
bool is_palindrome(const char s[], int n)
    // s 指向一个包含 n 个字符的数组的首字符
{
    int first = 0;           // 首字符的索引
    int last = n - 1;        // 尾字符的索引
    while (first < last) {   // 我们还未到达中央
        if (s[first] != s[last]) return false;
        ++first;              // 向前移动
        --last;                // 向后移动
    }
    return true;
}
```

为了测试 `is_palindrome()`，我们首先需要将字符读入数组。为了实现这一操作，一种安全的（即没有数组溢出危险的）方法如下：

```
istream& read_word(istream& is, char* buffer, int max)
    // 从 is 读取最多 max-1 个字符存入 buffer
{
    is.width(max);          // 下一个 >> 最多读取 max-1 个字符
    is >> buffer;           // 读取空白符间隔的单词，在将最后一个读入字符存入 buffer 后添加一个 0
    return is;
}
```

恰当地设置 `istream` 的宽度能避免下一个 `>>` 操作导致缓冲区溢出。不幸的是，这也意味着我们不知道读操作是遇见空白符结束的，还是缓冲区满结束的（因此还需要继续读入更多的字符）。另一方面，谁又能记得 `width()` 作用于输入流时的具体行为呢？标准库中 `string` 和 `vector` 更适合于作为输入缓冲区，因为它们能够根据输入的数据量动态调整大小。结尾 0 是必需的，因为对字符数组（C 风格字符串）的大多数常用操作都假设字符串以 0 结束。借助 `read_word()`，我们可以编写如下代码：

```
int main()
{
    constexpr int max = 128;
    for (char s[max]; read_word(cin, s, max); ) {
        cout << s << " is";
        if (!is_palindrome(s, strlen(s))) cout << " not";
        cout << " a palindrome\n";
    }
}
```

调用 `strlen(s)` 返回当调用 `read_word()` 结束之后数组中的字符数，`cout << s` 将输出数组中的字符，直至遇见字符 0 为止。

 我们认为这种“数组方法”比“`string` 方法”复杂得多，并且当我们尝试认真处理长字符串时，情况会变得更糟。参见习题 10。

13.7.3 使用指针实现回文

除了使用索引，我们还可以通过指针访问字符：

```
bool is_palindrome(const char* first, const char* last)
    // first 指向首字符, last 指向尾字符
{
    while (first < last) {           // 我们尚未到达中央
        if (*first != *last) return false;
        ++first;                      // 向前移动
        --last;                       // 向后移动
    }
    return true;
}
```

注意，我们确实可以对指针进行自增或自减操作，自增操作使指针指向数组中的下一个元素，而自减操作使指针指向上一个元素。如果指针指向的区域超出了数组的实际范围，那么将会产生严重的越界错误。这是使用指针可能会产生的另一个问题。

我们像下面这样调用 `is_palindrome()`：

```
int main()
{
    const int max = 128;
    for (char s[max]; read_word(cin, s, max); ) {
        cout << s << " is ";
        if (!is_palindrome(&s[0], &s[strlen(s)-1])) cout << " not ";
        cout << " a palindrome\n";
    }
}
```

我们还可以按如下方式重写 `is_palindrome()`（只是好玩）：

```
bool is_palindrome(const char* first, const char* last)
    // first 指向首字符, last 指向尾字符
{
    if (first < last) {
        if (*first != *last) return false;
        return is_palindrome(first+1, last-1);
    }
    return true;
}
```

当我们重新描述回文的定义时，上述代码的意义就显而易见了：一个单词是回文的充要条件是，其首字符与尾字符相同，且删除首尾字符所得子串仍然是回文。

简单练习

本章设置了两组练习：一组针对数组，另一组针对 `vector`，方式大致相同。读者应完成两组练习，并进行对比。

数组练习：

1. 定义大小为 10 的 `int` 类型的全局数组 `ga`，并将数组元素初始化为 1, 2, 4, 8, 16 等。
2. 定义具有两个参数的函数 `f()`，其中一个参数为 `int` 类型数组，另一个参数指出该数组包含的元素数量。
3. 在 `f()` 中：

- a. 定义大小为 10 的 int 类型的局部数组 `la`。
 - b. 将 `ga` 中的值拷贝至 `la`。
 - c. 打印 `la` 的所有元素
 - d. 定义 int 指针 `p`, 并初始化它指向一个在自由空间中分配的数组, 该数组与参数数组大小相同。
 - e. 将参数数组中的值拷贝至自由空间中的数组。
 - f. 打印自由空间中数组的所有元素。
 - g. 释放自由空间中的数组。
4. 在 `main()` 中:
- a. 调用 `f()` 并以 `ga` 为其参数。
 - b. 定义大小为 10 的数组 `aa`, 并将其元素初始化为前 10 个阶乘数 ($1, 2 \times 1, 3 \times 2 \times 1, 4 \times 3 \times 2 \times 1$, 等等)。
 - c. 调用 `f()` 并以 `aa` 为其参数。

标准库 `vector` 练习:

1. 定义全局变量 `vector<int>gv`, 并将其元素初始化为 10 个 int: $1, 2, 4, 8, 16$ 等。
2. 定义函数 `f()`, 接受一个 `vector<int>` 参数。
3. 在 `f()` 中:
 - a. 定义局部变量 `vector<int> lv`, 且 `lv` 的大小与参数 `vector` 相同。
 - b. 将 `gv` 的值拷贝至 `lv`。
 - c. 打印 `lv` 的所有元素。
 - d. 定义局部变量 `vector<int> lv2`, 并将其初始化为 `vector` 参数的副本。
 - e. 打印 `lv2` 的所有元素。
4. 在 `main()` 中:
 - a. 调用 `f()` 并以 `gv` 为其参数。
 - b. 定义向量 `vector<int> vv`, 并将其元素初始化为前 10 个阶乘数 ($1, 2 \times 1, 3 \times 2 \times 1, 4 \times 3 \times 2 \times 1$, 等等)。
 - c. 调用 `f()` 并以 `vv` 为其参数。

思考题

1. “买者自慎”的含义是什么?
2. 对于类对象而言, 拷贝的默认含义是什么?
3. 类对象拷贝的默认含义在什么情况下是合适的? 什么情况下是不合适的?
4. 什么是拷贝构造函数?
5. 什么是拷贝赋值?
6. 拷贝赋值与拷贝初始化之间有什么区别?
7. 什么是浅拷贝? 什么是深拷贝?
8. 一个 `vector` 对象的副本与该 `vector` 对象之间有何不同?
9. 类的五种必要操作有哪些?
10. 什么是显式构造函数? 在什么情况下应该使用显式构造函数?
11. 对于一个类对象而言, 哪些操作是被隐式调用的?

12. 什么是数组？
13. 如何拷贝一个数组？
14. 如何对数组初始化？
15. 什么时候应该使用指针参数而不是引用参数？为什么？
16. 什么是 C 风格字符串？
17. 什么是回文？

术语

array (数组)	essential operations (必要操作)
array initialization (数组初始化)	explicit constructor (<code>explicit</code> 构造函数)
copy assignment (拷贝赋值)	move assignment (移动赋值)
copy constructor (拷贝构造函数)	move construction (移动构造)
deep copy (深拷贝)	palindrome (回文)
default constructor (默认构造函数)	shallow copy (浅拷贝)

习题

1. 编写函数 `char *strdup(const char *)`，该函数能够将 C 风格字符串复制到其在自由空间上分配的内存中。要求：不使用任何标准库函数。使用解引用运算符 * 代替下标操作。
2. 编写函数 `char *findx(const char *s, const char *x)`，该函数能够在 C 风格字符串 s 中定位字符串 x 首次出现的位置。要求：不使用任何标准库函数。使用解引用运算符 * 代替下标操作。
3. 编写函数 `int strcmp(const char *s1, const char *s2)`，该函数能够比较 C 风格字符串。如果 s1 按照字典顺序排在 s2 之前，函数返回负整数；如果 s1 与 s2 相同，函数返回 0；如果 s1 按照字典顺序排在 s2 之后，函数返回正整数。要求：不使用任何标准库函数。使用解引用运算符 * 代替下标操作。
4. 考虑如下问题：如果向函数 `strdup()`、`findx()` 和 `strcmp()` 传递非 C 风格字符串的参数，会出现什么结果？试一试。首先，尝试向函数传递不以 0 结尾的字符数组（不要在实际代码（即非实验性代码）中编写这样的代码，否则可能会造成严重的破坏）。尝试传递在自由空间中及栈中分配的“虚假 C 风格字符串”。如果程序结果看上去仍然是合理的，那么请关闭 debug 模式。重新设计并实现这三个函数，使得这些函数接受另一个参数，该参数指出了字符串参数中能保存的最大元素个数。然后，用正确的 C 风格字符串以及“坏”字符串测试这些函数。
5. 编写函数 `string cat_dot(const string &s1, const string &s2)`，该函数连接参数字符串，中间以圆点相隔。例如，`cat_dot("Niels", "Bohr")` 返回结果 `Niels.Bohr`。
6. 修改上一题的 `cat_dot()`，使函数接受第三个字符串参数，用该字符串（而非圆点）作为分隔符。
7. 修改上一题的 `cat_dot()`，使函数以 C 风格字符串为参数，并以自由空间中分配的字符串作为返回结果。要求：不使用任何标准库函数或类型。保证所有在自由空间中（通过 `new`）分配的内存都被正确地释放（通过 `delete`）。比较完成此题所做的工作与完成习题 5、6 所做的工作。

8. 修改 13.6 节中所有函数，令它们通过构造单词的逆序副本并将副本与原单词比较的方式判定单词是否为回文。例如，接受参数 "home"，产生 "emoh"，然后比较两个字符串发现它们是不同的，因此 home 不是一个回文。
9. 考虑 12.3 节中的内存布局。编写一个程序，该程序能够指出静态存储、栈和自由空间在内存中的布局顺序。栈扩展的方向：是向高地址空间扩展还是向低地址空间扩展？在自由空间上分配的数组内，具有较高下标的元素是在高地址空间还是低地址空间？
10. 回顾 13.6.2 节中关于回文问题的数组解决方案。修改该方案使之能够对长字符串进行处理：(a) 当输入字符串过长时，会提示出错报告；(b) 能够处理任意长度的字符串。评价这两种实现版本的复杂度。
11. 查找（例如通过互联网）跳表（skip list）的相关知识并实现这种链表。此题有相当难度。
12. 编程实现游戏“猎杀怪兽”。“猎杀怪兽”（或称“怪兽”）是一个由 Gregory Yob 发明的简单（非图形化）电脑游戏。它的基本假设是一个满身臭味的怪物居住在一个由多间相连房间构成的黑暗山洞里。你的任务是用弓箭杀死怪兽。除了怪兽之外，山洞中还存在两种危险：无底陷阱和巨型蝙蝠。如果你进入的房间有无底陷阱，那么游戏将结束。如果房间中有蝙蝠，那么蝙蝠将抓住你，扔入另一房间之中。如果你进入了怪兽所在的房间或者怪兽进入了你所在的房间，它会吃掉你。当你进入一个房间时，若附近有危险，你将会得到提示：

“我闻到了怪兽的味道”：怪兽在相邻的房间中。

“我感到一阵微风吹来”：相邻的一间房间中有陷阱。

“我听到蝙蝠的声音”：相邻的房间中有蝙蝠。

山洞中的每一个房间均被编了号。每一个房间通过地道与其他三个房间相连。当进入一个房间时，你将会得到诸如“你在房间 12 中；房间 12 通过地道与房间 1、13、4 相连；移动还是射击？”一种可能的答案是 m13（代表移动到房间 13）以及 s13-4-3（向房间 13、4、3 射箭）。一支箭的覆盖范围为三个房间。在游戏开始时，你有五支箭。射击的后果是会惊醒怪兽，怪兽将会向相邻的房间逃跑——可能是你所在的房间。

这一练习的难点可能在于生成山洞的过程中决定房间的相连关系。你可能需要使用随机数发生器（例如 std_lib_facilities.h 中声明的 randint()），来使得每次运行程序都会产生不同的山洞以及随机移动蝙蝠和怪兽。提示：在调试中，应使程序能够产生关于山洞状态的输出。

附言

标准库 `vector` 建立在低层内存管理特性之上，例如指针与数组，而它的主要功能是帮助我们避免使用这些工具所带来的复杂性。当设计一个类时，我们必须考虑类的初始化、拷贝与析构。

向量、模板和异常

成功从不是终点。

——Winston Churchill

这一章将讨论最常见、最有用的 STL 容器 `vector` 的设计与实现。在本章中，我们将展示如何实现元素数量可变的容器，如何以参数形式指定容器中元素的类型，以及如何处理越界错误。与之前类似，本章中介绍的技术是通用的，而不仅仅局限于 `vector` 的实现，甚至不仅仅局限于容器的实现。对于各种不同的数据类型，我们将展示如何安全地处理数量可变的数据。此外，我们还增加了一些现实的设计案例。本章中介绍的技术依赖于模板与异常，所以我们将介绍如何定义模板，并介绍一些用于资源管理的基本技术，这是用好异常的关键。

14.1 问题

到第 13 章结束时，`vector` 已达到如下程度，我们可以：

- 创建任意数量双精度浮点数的 `vector` (`vector` 类对象)。
- 通过赋值与初始化拷贝 `vector` 对象。
- 依赖 `vector` 自身在其离开作用域时正确地释放占用的内存空间。
- 通过传统的下标操作访问 `vector` 的元素（既可以在赋值运算符左侧，也可以在右侧）。

已实现的这些特性都很好，也很有用，但为了达到我们所期望的高度（基于我们使用标准库 `vector` 的经验），我们需要解决下面三个问题：

- 如何改变 `vector` 对象的大小（改变元素数量）？
- 如何捕获并报告越界的 `vector` 元素访问？
- 如何用参数指定 `vector` 元素的类型？

例如，如何定义 `vector` 使得下面的代码是合法的：

```
vector<double> vd;           // double 类型元素
for (double d; cin>>d; )
    vd.push_back(d);         // 扩展 vd 以容纳所有元素
vector<char> vc(100);        // char 类型元素
int n;
cin>>n;
vc.resize(n);                // 令 vc 有 n 个元素
```

显然，令 `vector` 支持上述操作是十分有用的，但为什么从编程角度它们也很重要呢？对于想学习有用的编程技术以备后用的人来说，为什么它们很有趣呢？我们正在利用两方面的灵活性。我们有一个单一实体 `vector`，可以改变它的两方面属性：

- 元素的数量；
- 元素的类型。

这种可变性是最根本的非常有用的特性。在日常生活中，我们总是要收集数据。看看我的桌子，我看见过一堆银行对账单、信用卡账单以及电话话费单。而这些东西本质上是一系

列的各种类型的数据信息的集合：主要是字符串以及数值。在我的面前是一部电话，它记录了联系人的姓名与电话号码。在房间的书架上，满是书籍。我们的程序也是相似的：程序中有包含各种不同类型元素的容器。有很多不同种类的容器可供我们使用（`vector` 仅仅是最常用的一种），它们能存储诸如电话号码、姓名、交易额以及文档等信息。本质上来说，我桌子上和房间里的所有例子可以由某个计算机程序进行表示。明显的一个例外是电话：它本身就是一台计算机，当我们查找号码时，实际上看到的是一个程序的输出，就和我们现在编写的程序一样。实际上，这些号码可能被很好地保存在一个 `vector<Number>` 中。

显然，并不是所有容器都具有相同的元素数量。那么我们是否可以只使用大小在初始化时就固定下来的 `vector` 呢？也就是说，我们是否可以编写没有 `push_back()`、`resize()` 之类操作的代码呢？我们当然可以这么做，但这会带给程序员不必要的负担：在程序中只使用固定大小容器的基本技巧是，当元素数目增长到超出容器初始大小时，我们需要将元素移至一个更大的容器之中。例如，我们可以像下面代码一样读取输入存入 `vector` 中，而又从不改变 `vector` 的大小：

```
// 读取元素存入一个 vector 中，不使用 push_back:
vector<double>* p = new vector<double>(10);
int n = 0;           // 元素数目
for (double d; cin>>d; ) {
    if (n==p->size()) {
        vector<double>* q = new vector<double>(p->size()*2);
        copy(p->begin(), p->end(), q->begin());
        delete p;
        p = q;
    }
    (*p)[n] = d;
    ++n;
}
```

这段代码并不好。你相信我们已经正确实现了代码吗？你确认吗？注意我们是如何突然开始使用指针和显式内存管理的。我们所做的就是模仿当“接近机器”时必须使用的编程风格，只使用处理固定大小对象（数组，参见 13.6 节）的基本内存管理技术。然而，使用容器（如 `vector`）的一个重要的原因就是要比这做得更好；即，我们希望 `vector` 内部就能处理这种大小改变，从而帮助我们（它的用户）避免麻烦，减少出错的机会。换句话说，我们希望容器能够根据用户的实际需要动态调整其大小。例如：

```
vector<double> vd;
for (double d; cin>>d; ) vd.push_back(d);
```

 这种改变大小的操作是否经常发生呢？如果不是，那么改变大小的特性只是带来微小便利而已。然而，这种改变大小的操作在实际中是非常常见的。最明显的例子是从输入读取未知数量的数值。其他例子包括收集一次搜索操作的结果（我们事先不知道结果的总数）以及从集合中依次删除元素。因此，问题不是我们是否应该处理容器大小的变化，而是如何处理。

 我们到底为什么要为改变大小这种事烦恼呢？为什么不“一次性分配足够的内存供随后使用”呢？这看起来是一种最简单、最有效的策略。但是，只有当我们确实能分配足够的内存而又不会浪费大量空间时这种方法才适用，而我们实际上办不到。尝试采用这种方法的人将会不得不重写代码（如果他们仔细地、系统地检查了溢出的话），或是应付灾难性的结果

(如果他们没有仔细检查的话)。

显然，并不是所有 `vector` 都保存相同类型的元素。我们需要用 `vector` 保存 `double` 值、温度数据、记录（各种类型）、字符串、操作、GUI 按钮、形状、日期、窗口指针等等。可能性是无限的。

容器的类型多种多样。这是很重要的一点，而且它有很重要的隐含意义，因此我们不能不加思考地简单接受它。为什么不能只有 `vector` 一种容器？如果我们可以只使用一种容器（如 `vector`），我们就只需要将所有精力用于实现该容器，并可以将它作为编程语言的一部分。如果我们可以只使用一种容器，我们就不必费心学习其他类型的容器，只需一直使用 `vector` 即可。

数据结构对大多数重要应用都是十分关键的。大量厚重有用的书籍介绍了如何组织数据，而很多这类知识都可以回答这样一个问题：“我怎样存储自己的数据才是最好的？”答案就是我们需要很多不同类型的容器，但这一主题太大了，难以在本书中充分展开。不过，我们已经使用过 `vector` 和 `string` (`string` 是存储字符的容器)。在接下来的几章中，我们还将学习 `list`、`map` (`map` 是值对构成的树) 以及矩阵等容器。由于我们需要很多不同类型的容器，因此就需要学习已被广泛应用的用于构建和使用容器的语言特性和编程技术。实际上，我们用来保存和访问数据的技术对所有重要计算模型来说都是最为基础也最为有用的技术。

在内存管理的最底层，所有的对象都是固定大小并且不存在类型的概念。本章中介绍的语言特性和编程技术能令我们实现可变类型对象的容器，还能实现元素数目的改变。这能带给我们最根本的灵活性和便利性。

14.2 改变大小

为实现改变大小的目的，标准库 `vector` 采用了什么方法呢？它提供了三种简单的操作。假如我们定义了

```
vector<double> v(n); // v.size() == n
```

则可通过三种方法改变 `v` 的大小：

```
v.resize(10); // v 现在有 10 个元素
```

```
v.push_back(7); // 在 v 的末尾增加一个值为 7 的元素  
// v.size() 递增 1
```

```
v = v2; // 赋值为另一个 vector: v 现在变为 v2 的一个副本  
// v.size() 现在等于 v2.size()
```

标准库 `vector` 提供了更多可以改变自身大小的操作，如 `erase()` 和 `insert()`（见附录 C.4.7），但在本章中我们只考虑如何为 `vector` 实现上述三种操作。

14.2.1 表示方式

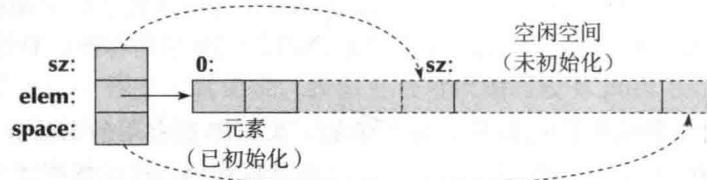
在 14.1 节中，我们展示了实现容器大小改变的最简单的策略：为新的元素数量分配内存空间，并将原有元素拷贝至新的空间中。然而，如果我们需要经常调整大小，那么这一策略非常低效。在实际中，如果我们改变了容器的大小，通常随后还会多次改变。特别是，我们很少只进行一次 `push_back()`。因此，我们可以针对这种预期来优化程序。实际上，所有 `vector` 实现都会记录元素数目和为“未来扩展”预留的“空闲空间”量。例如：

```

class vector {
    int sz;           // 元素数目
    double* elem;    // 首元素地址
    int space;       // 元素数加上用于新元素的（“当前分配的”）“空闲空间” / “槽位数”
public:
    ...
};

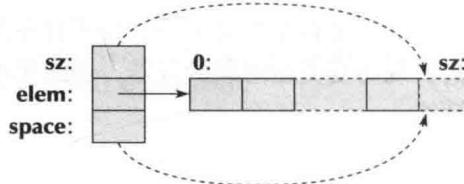
```

上述内容可图示如下：



由于我们从 0 开始为元素计数，因此 `sz`（元素数量）指向最后一个元素之后的位置，而 `space` 指向最后一个已分配单元之后的位置。图中的指针实际指示的是 `elem+sz` 和 `elem+space`。

当最初构造一个 `vector` 对象时，`space==sz`，即没有“空闲空间”：

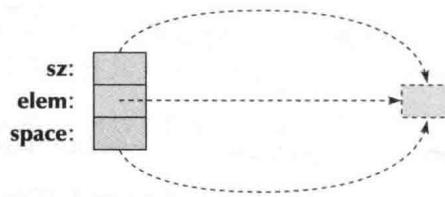


我们不会分配额外的空间，直到我们开始改变元素数目为止。一般而言，`space==sz`，因此没有额外的内存开销，除非我们使用 `push_back()`。

默认构造函数（创建一个空 `vector`）将整数成员设置为 0，将指针成员设置为 `nullptr`：

```
vector::vector() :sz{0}, elem{nullptr}, space{0} {}
```

效果如下图所示：



尾后元素完全是想象中的。默认构造函数不在空闲空间中分配内存，只占用最小的存储空间（但请参见习题 16）。

请注意，`vector` 所采用的技术可用于实现标准库 `vector`（及其他数据结构），但标准库的实现有相当的自由度，因此你的系统中的 `std::vector` 可能采用了不同的实现技术。

14.2.2 `reserve` 和 `capacity`

用于改变大小（即改变元素数量）的最基本的操作是 `vector::reserve()`。这一操作用来为新元素增加内存空间：

```

void vector::reserve(int newalloc)
{
    if (newalloc<=space) return; // 永远不会减少分配的空间
    double* p = new double[newalloc]; // 分配新空间
    for (int i=0; i<sz; ++i) p[i] = elem[i]; // 拷贝现有元素
    delete[] elem; // 释放旧空间
    elem = p;
    space = newalloc;
}

```

注意，我们并不对预留空间中的元素进行初始化。毕竟，我们只是预留空间以备将来使用，使用这些空间是 `push_back()` 和 `resize()` 的工作。

显然，用户可能关心 `vector` 对象中空闲空间的大小，因此我们（与标准库 `vector` 类似）提供了一个函数以获得这一信息：

```
int vector::capacity() const { return space; }
```

即，对于一个名为 `v` 的 `vector`，`v.capacity() - v.size()` 表示在不重新分配空间的前提下，我们用 `push_back()` 能够向 `v` 添加的元素数量。

14.2.3 resize

实现了 `reserve()` 后，再为 `vector` 实现 `reszie()` 就很简单了。我们只需处理以下几种情况：

- 新的大小大于已分配的空间。
- 新的大小大于当前大小，但小于或等于已分配空间。
- 新的大小等于当前大小。
- 新的大小小于当前大小。

下面的代码展示了 `resize()` 的实现：

```

void vector::resize(int newsize)
    // 令 vector 有 newsize 个元素
    // 用默认值 0.0 初始化每个新元素
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i) elem[i] = 0; // 初始化新元素
    sz = newsize;
}

```

我们用 `reserve()` 处理困难的内存空间管理问题。代码中的循环将初始化新的元素（如果有的话）。

在本例中，我们没有显式地处理每一种情况，但你可以验证：在上述代码中，每一种情况均被正确地处理了。

试一试

如果我们需要证明上述 `resize()` 是否正确，那么需要考虑（并测试）哪些情况？当 `newsize == 0` 时会怎样？当 `newsize == -77` 呢？

14.2.4 push_back

当我们刚开始思考实现 `push_back()` 时，它看起来有些复杂，但当实现了 `reserve()` 之

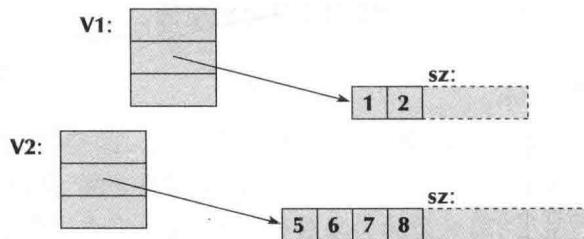
后，实现 `push_back()` 就变得相当简单了：

```
void vector::push_back(double d)
    // 将 vector 的大小增加 1；用 d 初始化新元素
{
    if (space==0)
        reserve(8);           // 从 8 个元素开始
    else if (sz==space)
        reserve(2*space);   // 获得更多空间
    elem[sz] = d;           // 将 d 添加到末尾
    ++sz;                  // 增加大小 (sz 为元素数)
}
```

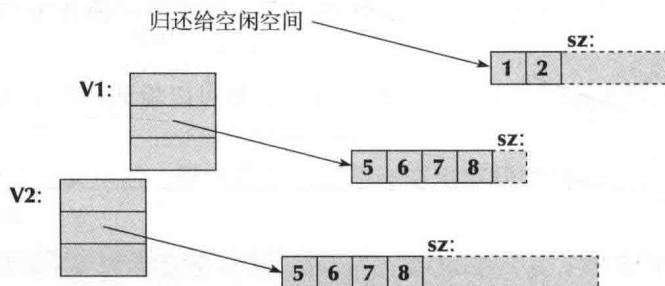
换句话说，如果没有空闲空间，我们将分配的空间大小加倍。实践已证明这种内存空间扩张策略对于绝大多数 `vector` 应用都是一个很好的选择，并且这种策略已被用于大多数标准库 `vector` 的实现。

14.2.5 赋值

我们可以用几种不同的方法实现向量的赋值。例如，我们可以仅在涉及的两个向量具有相同数量的元素时才判定赋值是合法的。但是，在 13.3.2 节中，我们决定向量赋值应具有更为通用的可能也是最为明显的意义：当赋值 `v1 = v2` 完成后，向量 `v1` 应该是向量 `v2` 的一个副本。例如：



显然，我们需要拷贝元素，那么空闲空间怎么处理呢？我们是否“拷贝”尾部的“空闲空间”？答案是否定的：新的 `vector` 将会获得元素的副本，但我们完全不了解新 `vector` 将被如何使用，因此我们无须为尾部的空闲空间操心：



最简单的实现包括如下操作：

- 为副本分配存储空间。
- 拷贝元素。
- 释放原有已分配的空间。

- 将 sz、elem、space 设置为新值。

如下所示：

```
vector& vector::operator=(const vector& a)
{
    // 类似拷贝构造函数，但我们必须处理原有元素
    {
        double* p = new double[a.sz];           // 分配新空间
        for (int i = 0; i < a.sz; ++i) p[i] = a.elem[i]; // 拷贝元素
        delete[] elem;                         // 释放旧空间
        space = sz = a.sz;                     // 设置新大小
        elem = p;                             // 设置新元素指针
        return *this;                          // 返回自引用
    }
}
```

作为惯例，赋值运算符将返回被赋值对象的引用。符号 `*this` 的含义参见 12.10 节。

上述实现是正确的，但通过观察我们可以发现上述实现包含了大量多余的存储空间分配和释放操作。如果被赋值 `vector` 的大小大于赋值对象会如何？如果被赋值 `vector` 的大小等于赋值对象会如何？在很多应用中，最后一种情况是十分常见的。在两种情况下，我们都只需将元素拷贝至目标 `vector` 中已就绪的新内存空间：

```
vector& vector::operator=(const vector& a)
{
    if (this == &a) return *this; // 自赋值，什么也不需要做

    if (a.sz <= space) {          // 空间足够，无须分配新空间
        for (int i = 0; i < a.sz; ++i) elem[i] = a.elem[i]; // 拷贝元素
        sz = a.sz;
        return *this;
    }

    double* p = new double[a.sz];           // 分配新空间
    for (int i = 0; i < a.sz; ++i) p[i] = a.elem[i]; // 拷贝元素
    delete[] elem;                         // 释放旧空间
    space = sz = a.sz;                     // 设置新大小
    elem = p;                             // 设置新元素指针
    return *this;                          // 返回自引用
}
```

在本例中，我们首先检测自引用（如 `v=v`）；在这种情况下，我们不需要做任何事情。这一检测在逻辑上看是多余的，但有时会带来明显的性能优化。这里展示了 `this` 指针的一种常见用途，检测参数 `a` 与调用成员函数（本例中是 `operator=()`）的对象是否是同一个对象。请确认，如果我们删除了 `this == &a` 这一行，代码仍然能够正确工作。另外，`a.sz <= space` 也是一处优化。请确认，如果我们删除了 `a.sz <= space`，代码仍然能够正确工作。

14.2.6 到目前为止的 `vector` 类

现在我们已经有了一个接近实用的 `double` 元素的 `vector`：

```
// 接近实用的 double 的 vector
class vector {
/*
    不变式：
    若 0 <= n < sz, elem[n] 为编号为 n 的元素
    sz <= space;
    若 sz < space，则在 elem[sz-1] 之后有能容纳 (space - sz) 个 double 的空间
*/}
```

```

int sz;           // 大小
double* elem;   // 指向元素的指针（或 0）
int space;      // 元素数加上空闲位置数
public:
vector() : sz{0}, elem{nullptr}, space{0} { }
explicit vector(int s) : sz{s}, elem{new double[s]}, space{s}
{
    for (int i=0; i<sz; ++i) elem[i]=0;    // 元素被初始化
}

vector(const vector&);           // 拷贝构造函数
vector& operator=(const vector&); // 拷贝赋值

vector(vector&&);             // 移动构造函数
vector& operator=(vector&&);  // 移动赋值
~vector() { delete[] elem; }     // 析构函数

double& operator[ ](int n) { return elem[n]; } // 访问：返回引用
const double& operator[ ](int n) const { return elem[n]; }

int size() const { return sz; }
int capacity() const { return space; }

void resize(int newsize);         // 增长
void push_back(double d);
void reserve(int newalloc);
};

```

请注意上述代码是如何实现那些必要操作的（见 13.4 节）：构造函数、默认构造函数、拷贝操作、析构函数。此 `vector` 实现了元素访问操作（下标`[]`），实现了获取数据信息的操作（`size()` 与 `capacity()`），还实现了控制大小变化的操作（`resize()`、`push_back()` 和 `reserve()`）。

14.3 模板

但是，我们不只需要 `double` 的 `vector`，我们希望能够自由地指定 `vector` 的元素类型。例如：

```

vector<double>
vector<int>
vector<Month>
vector<Window*>           // Window 指针的 vector
vector<vector<Record>>    // Record 的 vector
vector<char>

```

为了达到这一目的，我们必须知道如何定义模板。本书从最初就开始使用模板了，但到目前为止我们还未曾定义过一个模板。标准库为我们提供了迄今为止所需要的特性，但我们不能简单地相信魔法，而需要清楚标准库的设计者和实现者是如何提供像 `vector` 这样的类型和 `sort()` 这样的函数的（见 16.1 节和附录 C.5.4）。这不仅有理论上的意义，还有很重要的编程实践意义，因为通常标准库所采用的工具、技术对我们编写自己的代码是非常有用的。例如，在第 16 和 22 章中，我们将展示模板是如何用于实现标准库容器和算法的。在第 24 章中，我们将展示如何为科学计算设计矩阵类型。

本质上说，模板是一种机制，它令程序员能够使用类型作为类或函数的参数。当随后我们提供具体类型作为参数时，编译器会为之生成特定的类或函数。

14.3.1 类型作为模板参数

我们希望令元素类型成为 vector 的参数。因此，我们将 vector 中的 double 用 T 代替——T 是一个参数，能被赋予 double、int、string、vector<Record> 和 Window* 之类的“值”。在 C++ 中引入类型参数 T 的语法为 template<typename T> 前缀，其含义是“对所有类型 T”。例如：

```
// 接近实用的类型 T 的 vector:
template<typename T>
class vector {           // 读作“对所有类型 T”(就像在数学中那样)
    int sz;                // 大小
    T* elem;               // 指向元素的指针
    int space;              // 大小 + 空闲空间
public:
    vector() : sz{0}, elem{nullptr}, space{0} {}
    explicit vector(int s) : sz{s}, elem{new T[s]}, space{s}
    {
        for (int i=0; i<sz; ++i) elem[i]=0;      // 元素被初始化
    }

    vector(const vector&);          // 拷贝构造函数
    vector& operator=(const vector&); // 拷贝赋值

    vector(vector&&);            // 移动构造函数
    vector& operator=(vector&&); // 移动赋值

    ~vector() { delete[] elem; }     // 析构函数

    T& operator[](int n) { return elem[n]; } // 访问：返回引用
    const T& operator[](int n) const { return elem[n]; }

    int size() const { return sz; }      // 当前大小
    int capacity() const { return space; }

    void resize(int newsize);          // 增长
    void push_back(const T& d);
    void reserve(int newalloc);
};
```

这个定义就是将 14.2.6 节中的 double 值 vector 中的 double 替换为模板参数 T 而得到的。我们可以像下面这样使用类模板 vector：

```
vector<double> vd;           // T 为 double
vector<int> vi;               // T 为 int
vector<double*> vpd;         // T 为 double*
vector<vector<int>> vvi;     // T 为 vector<int>, 其中 T 为 int
```

当我们使用模板时，可以认为编译器是按照如下方式生成类的：用实际类型（模板实参）取代模板参数。例如，当编译器遇到代码中的 vector<char> 时，它将（在某处）生成如下代码：

```
class vector_char {
    int sz;                  // 大小
    char* elem;               // 指向元素的指针
    int space;                // 大小 + 空闲空间
public:
    vector() : sz{0}, elem{nullptr}, space{0} {}
    explicit vector_char(int s) : sz{s}, elem{new char[s]}, space{s}
    {
        for (int i=0; i<sz; ++i) elem[i]=0;      // 元素被初始化
```

```

}

vector_char(const vector_char&);           // 拷贝构造函数
vector_char& operator=(const vector_char&); // 拷贝赋值

vector_char(vector_char&&);               // 移动构造函数
vector_char& operator=(vector_char&&);    // 移动赋值

~vector_char();                           // 析构函数

char& operator[] (int n) { return elem[n]; } // 访问：返回引用
const char& operator[] (int n) const { return elem[n]; }

int size() const;                         // 当前大小
int capacity() const;

void resize(int newsize);                // 增长
void push_back(const char& d);
void reserve(int newalloc);
};

```

对于 `vector<double>`，编译器生成的大致就是 14.2.6 节中的（`double` 的）`vector`（使用一个合适的内部名字来表示 `vector<double>`）。

 我们有时称类模板为类型生成器（type generator），称由一个类模板按给定模板实参生成类型（类）的过程为特例化（specialization）或模板实例化（template instantiation）。例如，`vector<char>` 和 `vector<Poly_line*>` 被称为 `vector` 的特例化版本。对于简单的模板，如我们的 `vector`，实例化是一个相当简单的过程。但对于更通用、更复杂的模板，实例化是一个相当复杂的过程。幸运的是，模板实例化的复杂性是编译器设计者而不是模板使用者需要解决的问题。模板实例化（生成模板特例化版本）只占用程序的编译时间或链接时间，而不会占用程序运行时间。

自然，我们也可以使用类模板的成员函数。例如：

```

void fct(vector<string>& v)
{
    int n = v.size();
    v.push_back("Norah");
    // ...
}

```

当使用这种类模板成员函数时，编译器将生成适合的函数。例如，当编译器遇到 `v.push_back("Norah")` 时，它会根据模板定义

```
template<typename T> void vector<T>::push_back(const T& d) /* ... */;
```

生成函数

```
void vector<string>::push_back(const string& d) /* ... */
```

这样，就有了一个函数 `v.push_back("Norah")` 供调用了。换句话说，当你需要一个函数处理给定对象和实参类型时，编译器会根据模板为你生成一个函数。

你可以在模板中使用 `template< class T >` 代替 `template< typename T >`。两者完全相同，但有些人喜欢 `typename`，“因为它的含义更清楚”并且“因为没人会对 `typename` 困惑，没人会认为不能使用内置类型（如 `int`）作为模板实参”。我们认为 `class` 这一名称已经包含了类

型的含义，因此使用 `class` 并没有什么不好。而且，关键字 `class` 要更短些。

14.3.2 泛型编程

在 C++ 中，模板是泛型编程的基础。实际上，C++ 中“泛型编程”的定义就是“使用 ~~模板~~ 模板”，虽然这样的定义有点太简单化了。我们不应根据编程语言特性定义基本的编程概念。编程语言特性主要用于支持编程技术——而不是相反。和其他流行概念一样，“泛型编程”存在多种定义。我们认为简单的、最有用的定义是：

泛型编程 (generic programming)：编写能够正确处理以参数形式呈现的各种类型的代码，只要这些参数类型满足特定的语法和语义要求。

例如，`vector` 的元素必须是可拷贝的类型（通过拷贝构造和拷贝赋值）。在第 15 和 16 章中，我们将介绍要求其参数能进行算术运算的模板。当参数化的是一个类时，我们将得到一个类模板（class template），通常也称为参数化类型（parameterized type）或者参数化类（parameterized class）。当参数化的是一个函数时，我们将得到一个函数模板（function template），通常也称为参数化函数（parameterized function），有时也称为算法（algorithm）。因此，泛型编程有时称为“面向算法的程序设计”，设计重点在于算法而非算法所使用的数据类型。

由于参数化类型的概念是编程的核心，我们需要进一步探讨这个有些让人困惑的术语。这样，当我们在其他场合中再次碰到这一概念时，才有可能不对它太过困惑。

这种依赖于显式模板参数的泛型编程通常被称为参数化多态（parametric polymorphism）。相反，从类层次与虚函数获得的多态被称为即时多态（ad hoc polymorphism），而这种编程风格被称为面向对象编程（object-oriented programming，见 19.3 ~ 19.4 节）。之所以两类编程都被称为多态（polymorphism），是因为每种类型都依赖于程序员通过一个单一接口表示一个概念的多个版本。多态在希腊语中是“多种形状”的意思，这表示很多不同的类型，你可以通过一个公共接口操纵它们。在第 12~14 章及第 21 章的 `Shape` 例子中，我们可以通过 `Shape` 定义的接口访问多种形状（如 `Text`、`Circle` 和 `Polygon`）。当我们使用 `vector` 时，我们通过 `vector` 模板定义的接口使用多种 `vector`（如 `vector<int>`、`vector<double>` 和 `vector<Shape*>`）。

面向对象编程（使用类层次和虚函数）和泛型编程（使用模板）之间存在一些差异。最明显的差异是，当你使用泛型编程时，被调用函数的选择由编译器在编译时确定，而对于面向对象编程，被调用函数的选择直到运行时才确定。例如：

```
v.push_back(x);      // 将 x 放入 vector v
s.draw();           // 绘制形状 s
```

对于 `v.push_back(x)`，编译器将确定 `v` 的元素类型并使用对应的 `push_back()`；但对于 `s.draw()`，编译器将会间接调用某个 `draw()` 函数（使用 `s` 的 `vtbl`，参见 19.3.1 节）。这一差异令面向对象编程比泛型编程更自由，但普通的泛型编程更为规则，更容易理解，能被更好地执行（因此用“即时”和“参数化”进行区分）。

两种编程可总结如下：

- 泛型编程：由模板支撑，依赖编译时解析。
- 面向对象编程：由类层次和虚函数支撑，依赖运行时解析。

将这两种类型的编程相结合是可行的，也是有用的。例如：

```
void draw_all(vector<Shape*>& v)
{
    for (int i = 0; i < v.size(); ++i) v[i] -> draw();
}
```

在此代码中，我们对一个基类（`Shape`）调用了一个虚函数（`draw()`）——这当然是面向对象编程。但是，我们还将 `Shape*` 指针存放在一个 `vector` 中，`vector` 是一个参数化类型，因此我们也使用了（简单的）泛型编程。

假设你到现在已经看够了相关的思想和观念，那么，人们用模板来做什么？答案是，为了无与伦比的灵活性和性能，我们应该：

- 在对性能要求高的场合使用模板（例如，数值计算和强实时；参见第 24、25 章）。
- 在需要灵活组合不同类型信息的场合中使用模板（如 C++ 标准库；参见第 15、16 章）。

14.3.3 概念

模板有很多有用的特性，例如极大的灵活性和接近最优的性能，但不幸的是模板不是完美的。一如既往，优点伴随着缺点。主要问题是，获得高灵活性和性能的代价是模板的“内在”（定义）与其接口（声明）不能很好地分离。这令其错误诊断变得糟糕——通常只能得到相当糟糕的错误信息。有时，这些错误信息在编译过程中的出现时间远远落后于我们的期望。

当编译使用模板的代码时，编译器会“探查”模板内部以及模板实参。它这样做是为了获得生成优化代码所需的信息。为了获得这种信息，当今的编译器都要求在使用模板的地方必须能得到模板的完整定义。这包括调用的所有成员函数和所有模板函数。因此，模板的编写者会将模板定义放在头文件中。这并不是 C++ 标准所要求的，但在有根本改进的 C++ 编译器广泛普及之前，我建议你这样处理自己的模板：对于会在多个编译单元中使用的模板，将其定义放在一个头文件中。

这部分内容的学习应以编写简单模板开始，随后小心探索前进，获得更多经验。一种有用的技术是我们设计 `vector` 所采用的技术：首先设计一个针对特定类型的类并测试它。这个类设计好后，将特定类型替换为模板参数，并用不同的模板实参测试它。在实际编程中，应优先选用基于模板的库（如 C++ 标准库）来获得通用性、类型安全和高性能。第 15、16 章介绍标准库容器和算法，会向你展示模板的使用。

C++14 提供了一种机制，可极大地改善模板接口的检查。例如，我们如果在 C++11 中编写如下代码：

```
template<typename T>
class vector {
    // ...
};
```

那么无法准确陈述对实参类型 `T` 有什么期望。C++ 标准描述了对实参类型有什么要求，但只是用英语描述的，而不是用编译器能理解的代码描述的。我们称一个模板实参应满足的要求集合为概念（concept）。一个模板实参必须满足模板对它的要求，即概念。例如，`vector` 要求其元素能拷贝或移动、可获取地址且能默认构造（如果需要的话）。换句话说，元素必须满足一组要求——我们可以称之为 `Element`。在 C++14 中，我们可以显式陈述概念：

```
template<typename T> // 对所有类型 T
    requires Element<T>() // 对所有满足 Element 的类型 T
class vector {
    ...
};
```

这说明一个概念实际上是一个类型谓词，即一个编译时求值的（`constexpr`）函数，若类型实参（本例中的 `T`）具有概念所要求的属性（本例中的 `Element`），此函数返回 `true`，否则返回 `false`。这有些冗长，但我们可以使用如下简写语法：

```
template<Element T> // 对所有令 Element<T>() 为 true 的类型 T
class vector {
    ...
};
```

如果没有支持概念的 C++14 编译器，我们可以通过命名和注释来说明要求：

```
template<typename Elem> // 要求 Element<Elem>()
class vector {
    ...
};
```

编译器不理解我们的命名，也不会阅读我们的注释，但显式说明概念可以帮助我们思考代码，改进通用代码的设计，并帮助其他程序员理解我们的代码。在学习、使用泛型编程的过程中，我们可以使用一些常见的、有用的概念：

- `Element<E>()`: `E` 可以是容器的元素。
- `Container<C>()`: `C` 可以保存 `Element`，并能作为一个 `[begin():end()]` 序列访问。
- `Forward_iterator<For>()`: `For` 可以用来访问序列 `[b:e]` (如链表、向量或数组)。
- `Input_iterator<In>()`: `In` 可以用来读取序列 `[b:e]`，只能读取一次 (如输入流)。
- `Output_iterator<Out>()`: `Out` 可用来输出一个序列。
- `Random_access_iterator<Ran>()`: `Ran` 可以用来反复读取和写入序列 `[b:e]`，并支持下标操作 `[]`。
- `Allocator<A>()`: `A` 可以用来获取和释放内存 (例如自由存储空间)。
- `Equal_comparable<T>()`: 我们可以用 `==` 比较两个 `T` 是否相等，得到一个布尔结果。
- `Equal_comparable<T,U>()`: 我们可以用 `==` 比较一个 `T` 和一个 `U` 是否相等，得到一个布尔结果。
- `Predicate<P,T>()`: 我们可以用一个类型为 `T` 的实参调用 `P`，得到一个布尔结果。
- `Binary_predicate<P,T>()`: 我们可以用两个类型为 `T` 的实参调用 `P`，得到一个布尔结果。
- `Binary_predicate<P,T,U>()`: 我们可以用两个类型分别为 `T` 和 `U` 的实参调用 `P`，得到一个布尔结果。
- `Less_comparable<L,T>()`: 我们可以用 `L` 而非 `<` 比较两个 `T` 是否前者小于后者，得到一个布尔结果。
- `Less_comparable<T,U>()`: 我们可以用 `L` 而非 `<` 比较一个 `T` 是否小于一个 `U`，得到一个布尔结果。
- `Binary_operation<B,T>()`: 我们可以用 `B` 对两个 `T` 执行一个操作。
- `Binary_predicate<B,T,U>()`: 我们可以用 `B` 对一个 `T` 和一个 `U` 执行一个操作。
- `Number<N>()`: `N` 类似数字，支持 `+`、`-`、`*` 和 `/`。

对标准库容器和算法，这些（和更多）概念的说明非常详细。在本书中，特别是第 15、16 章中，我们将用它们非正式地说明我们的容器和算法。

每个容器类型和迭代器类型 **T** 都有一个值类型（用 **Value_type<T>** 表示），对应元素类型。此 **Value_type<T>** 通常实现为类模板的成员类型 **T::value_type**；参见 **vector** 和 **list**（见 15.5 节）。

14.3.4 容器和继承

人们总是尝试一种不可行的面向对象编程和泛型编程的组合方式：试图将派生类对象的容器作为基类对象的容器使用。例如：

```
vector<Shape> vs;
vector<Circle> vc;
vs = vc;           // 错误：要求 vector<Shape>
void f(vector<Shape>&); // 错误：要求 vector<Shape>
f(vc);            // 错误：要求 vector<Shape>
```

 但为什么不行呢？毕竟你会说“我能够将一个 **Circle** 转换为一个 **Shape**”。但实际上你不能这么做。你可以将一个 **Circle*** 转换为一个 **Shape***，或者将一个 **Circle&** 转换为一个 **Shape&**，但我们已经有意地禁止了 **Shape** 对象之间的赋值，因此你不必担心将一个具有半径属性的 **Circle** 存入一个没有半径属性的 **Shape** 变量时将会发生什么（见 19.2.4 节）。假如我们允许这种赋值，将会发生什么呢？类对象将会发生“切片”现象，类似整数截断（见 3.9.2 节）。

因此，我们改为尝试使用指针：

```
vector<Shape*> vps;
vector<Circle*> vpc;
vps = vpc;           // 错误：要求 vector<Shape*>
void f(vector<Shape*>&); // 错误：要求 vector<Shape*>
f(vpc);            // 错误：要求 vector<Shape*>
```

这一次，系统仍然报错；为什么呢？看一看 **f()** 可能会做些什么：

```
void f(vector<Shape*>& v)
{
    v.push_back(new Rectangle{Point{0,0}, Point{100,100}});
}
```

显然，我们可以将一个 **Rectangle*** 指针存入一个 **vector<Shape*>** 中。但是，如果这一 **vector<Shape*>** 对象在别的地方被解释为一个 **vector<Circle*>** 时，可能会得到令人惊讶的糟糕结果。特别地，假设上述例子能够在编译器中成功编译，那么在 **vpc** 中存放 **Rectangle*** 指针会产生什么结果呢？继承是一种强大但微妙的机制，而模板并没有隐含地扩展它。存在几种用模板表达继承的方法，但这些内容不在本书的介绍范围之内。我们只需记住，对于任意模板 **C**，“**D** 是 **B**”并不意味着“**C<D>** 是 **C**”——并且应重视这一规则的价值，它能避免意外的类型违规。参见 25.4.4 节。

14.3.5 整数作为模板参数

显然，用类型来参数化类是很有用的。那么，用“其他东西”（如整型值或字符串值）来参数化类又会怎样呢？本质上，任何类别的实参都是有用的，但我们只考虑类型和整数作为参数。其他类别的参数并不像这两种参数那么有用，并且在 C++ 中使用其他类别的参数需

要掌握一些非常细节的语言特性。

下面我们讨论一个例子，这是整型值作为模板实参的最常见的用途——一个容器所包含的元素数在编译时就已确定：

```
template<typename T, int N> struct array {
    T elem[N];           // 在成员数组中保存元素

    // 依赖于默认构造函数、析构函数和赋值操作
    T& operator[] (int n);          // 访问：返回引用
    const T& operator[] (int n) const;

    T* data() { return elem; }      // 转换为 T*
    const T* data() const { return elem; }

    int size() const { return N; }
};
```

我们可以像下面这样使用 `array` (参见 15.7 节)：

```
array<int,256> gb;           // 256 个整数
array<double,6> ad = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5 };
const int max = 1024;

void some_fct(int n)
{
    array<char,max> loc;
    array<char,n> oops;        // 错误：编译器不知道 n 的值
    ...
    array<char,max> loc2 = loc; // 创建副本作为备份
    ...
    loc = loc2;                // 恢复数据
    ...
}
```

显然，`array` 是很简单的——比 `vector` 更简单，功能也更有限——那么人们为什么还要使用 `array` 而不是 `vector` 呢？一种答案是“效率”。我们在编译时就已经知道 `array` 的大小，因此编译器可以（为全局对象如 `gb`）分配静态内存和（为局部变量如 `loc`）分配栈内存而不是在自由存储空间中分配内存。当我们进行范围检查时，可直接与常量（大小参数 `N`）进行比较。对于大多数程序而言，效率的提高并不那么显著，但如果你编写的是一个关键的系统组件，例如网络驱动，即使很小的性能提升也很重要。更重要的是，有些程序可能不允许使用自由空间。这类程序通常是嵌入式系统程序和 / 或安全攸关的程序（参见第 25 章）。在这类程序中，`array` 比 `vector` 更具有优势，因为不会违反临界限制（不使用自由空间）。

让我们现在考虑一个相反的问题：不是“为什么不能简单地使用 `vector`？”而是“为什么不能简单地使用内置数组？”如我们在 13.5 节中所见，使用数组容易造成错误：数组不知道自身的大小，可以很容易地转换为指针，不能正确拷贝；而 `array` 则类似 `vector`，不存在这些问题。例如：

```
double* p = ad;           // 错误：不能隐式转换为指针
double* q = ad.data();     // 正确：显式转换

template<typename C> void printout(const C& c) // 函数模板
{
    for (int i = 0; i < c.size(); ++i) cout << c[i] << '\n';
}
```

与 `vector` 类似，我们可以对 `array` 调用 `printout()`：

```
printout(ad);           // 用 array 调用
vector<int> vi;
// ...
printout(vi);          // 用 vector 调用
```

这是一个将泛型编程应用于数据访问的简单例子。这段代码能够正确运行的原因在于 `array` 和 `vector` 的接口（`size()` 和下标操作）是相同的。第 15 和 16 章将会详细介绍这种编程风格。

14.3.6 模板实参推断

对于一个类模板，当你创建某个特定类的对象时，需要指定模板实参。例如：

```
array<char,1024> buf;      // 对 buf, T 是 char 且 N 是 1024
array<double,10> b2;       // 对 b2, T 是 double 且 N 是 10
```



对于函数模板，编译器通常能够根据函数实参推断出模板参数。例如：

```
template<class T, int N> void fill(array<T,N>& b, const T& val)
{
    for (int i = 0; i < N; ++i) b[i] = val;
}

void f()
{
    fill(buf,'x');        // 对 fill(), T 是 char 且 N 是 1024
                           // 因为这是 buf 所具有的参数
    fill(b2,0.0);         // 对 fill(), T 是 double 且 N 是 10
                           // 因为这是 b2 所具有的参数
}
```

在技术上，`fill(buf,'x')` 是 `fill<char,1024>(buf,'x')` 的简写，`fill(b2,0)` 是 `fill<double, 10>(b2, 0)` 的简写。幸运的是，我们通常并不需要编写这么具体的代码。编译器能够为我们做这些事情。

14.3.7 泛化 `vector`

当我们将 `vector` 从“`double` 的 `vector`”类泛化至“`T` 的 `vector`”模板时，我们并没有考虑 `push_back()`、`resize()` 和 `reserve()` 的定义。现在，我们必须重新考虑它们的定义，因为在 14.2.2 和 14.2.3 节中，这些函数的定义基于一些假设，这些假设对 `double` 是成立的，但并不是对所有其他元素类型都成立：

- 如果类型 `X` 没有默认值，我们应如何处理 `vector<X>`？
- 当元素使用完毕时，我们如何确保它们被销毁了？

我们必须解决这些问题吗？我们可能会说，“不要用不具有默认值的类型创建 `vector`”“将 `vector` 用于具有析构函数的类型时，使用方式不要引起问题”。但对于一个以“通用”为目标的模板而言，上述限制对用户是很恼人的，并且会给人这样的印象：设计者并没有仔细思考过这个问题或是根本不关心用户。通常，这种猜疑是正确的，而标准库的设计者就未留下这些问题。为了构造与标准库相同的 `vector`，我们必须解决这两个问题。

为了处理没有默认值的类型，我们可以设置一个用户选项，以便在我们需要一个“默认值”时能够指定使用什么值：

```
template<typename T> void vector<T>::resize(int newsize, T def = T());
```

即除非用户指定了其他值，否则使用 `T()` 作为默认值。例如：

```
vector<double> v1;
v1.resize(100);           // 添加 100 个 double() (即 0.0)
v1.resize(200, 0.0);      // 添加 100 个 0.0——指定 0.0 是冗余的
v1.resize(300, 1.0);      // 添加 100 个 1.0

struct No_default {
    No_default(int);        // No_default 唯一的构造函数
    ...
};

vector<No_default> v2(10);   // 错误：试图创建 10 个 No_default()
vector<No_default> v3;
v3.resize(100, No_default(2)); // 添加 100 个 No_default(2) 的副本
v3.resize(200);              // 错误：试图添加 100 个 No_default()
```

析构函数的问题要更难以解决。基本上，我们需要处理非常尴尬的情况：数据结构同时包含已初始化数据和未初始化数据。到目前为止，我们已经花了很长时间学习避免未初始化数据以及通常伴随它的编程错误。现在——作为 `vector` 的实现者——我们必须面对这一问题，从而令我们——在作为 `vector` 的用户时——不必在实际应用中处理这些问题。

首先，我们需要寻找一种获得并管理未初始化内存空间的方法。幸运的是，标准库为我们提供了 `allocator` 类，该类能够提供未初始化内存。下面代码给出了 `allocator` 的一个稍微简化的版本：

```
template<typename T> class allocator {
public:
    ...
    T* allocate(int n);           // 为 n 个类型为 T 的对象分配空间
    void deallocate(T* p, int n); // 释放从 p 开始的 n 个类型为 T 的对象

    void construct(T* p, const T& v); // 在地址 p 构造一个值为 v 的 T 类型对象
    void destroy(T* p);            // 销毁 p 中的 T
};
```

如果你想了解更多细节，请参考《The C++ Programming Language》中的 `<memory>`（见附录 C.1.1），或者参考 C++ 标准。但是，本节内容表明，我们能用这四个基本操作实现：

- 分配能够容纳一个 `T` 类型对象的未初始化的内存空间。
- 在未初始化空间中构造 `T` 类型对象。
- 销毁一个 `T` 类型对象，并将其内存重置为未初始化状态。
- 释放能够容纳一个 `T` 类型对象的未初始化内存空间。

`allocator` 正是我们实现 `vector<T>::reserve()` 需要使用的工具。我们先向 `vector` 传递一个分配器参数：

```
template<typename T, typename A = allocator<T>> class vector {
    A alloc;           // 用 alloc 管理元素内存
    ...
};
```

除了提供一个 `allocator` 并默认使用标准的分配器而不是使用 `new` 之外，一切与之前的版本完全相同。作为 `vector` 的用户，我们可以忽略分配器，直至发现我们需要 `vector` 能够按照一种不太一样的方式管理其元素占用的内存空间。作为 `vector` 的实现者以及试图理解基本问题并学习基本技术的学习者，我们必须明白 `vector` 是如何处理未初始化内存并为

其用户构造合适的对象的。唯一影响到的代码是 `vector` 中直接处理内存的成员函数，例如 `vector<T>::reserve()`：

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return; // 从不减少分配的空间
    T* p = alloc.allocate(newalloc); // 分配新空间
    for (int i=0; i<sz; ++i) alloc.construct(&p[i], elem[i]); // 拷贝
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]); // 销毁
    alloc.deallocate(elem, space); // 释放旧空间
    elem = p;
    space = newalloc;
}
```

我们在未初始化空间中构造副本来移动元素，然后销毁原有的元素。我们不能使用赋值，因为对于 `string` 这样的类型，赋值操作会假设目标空间已被初始化。

实现了 `reserve()` 之后，`vector<T, A>::push_back()` 就容易实现了：

```
template<typename T, typename A>
void vector<T,A>::push_back(const T& val)
{
    if (space==0) reserve(8); // 从 8 个元素的空间开始
    else if (sz==space) reserve(2*space); // 获取更多空间
    alloc.construct(&elem[sz], val); // 将 val 添加到末尾
    ++sz; // 增大大小
}
```

类似地，`vector<T,A>::resize()` 也不难实现：

```
template<typename T, typename A>
void vector<T,A>::resize(int newsize, T val = T())
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i) alloc.construct(&elem[i], val); // 构造
    for (int i = newsize; i<sz; ++i) alloc.destroy(&elem[i]); // 销毁
    sz = newsize;
}
```

注意，由于有些类型没有默认构造函数，因此我们再次提供了用户选项，可指定一个值作为新元素的初始值。

本例中另一个新内容是当我们缩小一个 `vector` 时析构“剩余元素”，我们可以将析构函数看作将一个有类型对象转换为“裸内存”的操作。

⚠ “摆弄分配器”属于非常高级的 C++ 特性，我们应该先将其放在一边，直到我们已经准备好向 C++ 专家迈进为止。

14.4 范围检查和异常

回顾目前的 `vector` 版本，我们会发现它没有对数据访问进行范围检查。`operator[]` 的实现十分简单：

```
template<typename T, typename A> T& vector<T,A>::operator[](int n)
{
    return elem[n];
}
```

那么，考虑下面代码：

```
vector<int> v(100);
v[-200] = v[200];      // 糟糕!
int i;
cin >> i;
v[i] = 999;             // 破坏一个随机的内存地址
```

上述代码能够编译成功并运行，但它访问了不属于我们的 `vector` 对象的内存空间。这可能会造成严重后果！在实际程序中，这样的代码是不可接受的。下面我们将完善 `vector` 以处理此问题。最简单的方法是增加一个名为 `at()` 的操作，可实现带范围检查的元素访问：

```
struct out_of_range /* ... */; // 此类用来报告越界访问错误

template<typename T, typename A = allocator<T>> class vector {
    ...
    T& at(int n);           // 带范围检查的访问
    const T& at(int n) const; // 带范围检查的访问
    T& operator[](int n);    // 不带检查的访问
    const T& operator[](int n) const; // 不带检查的访问
    ...
};

template<typename T, typename A > T& vector<T,A>::at(int n)
{
    if (n<0 || sz<=n) throw out_of_range();
    return elem[n];
}

template<typename T, typename A > T& vector<T,A>::operator[](int n)
// 照旧
{
    return elem[n];
}
```

有了 `at()`，我们可以编写如下代码：

```
void print_some(vector<int>& v)
{
    int i = -1;
    while(cin >> i && i != -1)
        try {
            cout << "v[" << i << "]==" << v.at(i) << "\n";
        }
        catch(out_of_range) {
            cout << "bad index: " << i << "\n";
        }
}
```

在这段代码中，我们通过 `at()` 进行带范围检查的数据访问，并且捕获 `out_of_range` 以避免非法的数据访问。

一般做法是，当我们确定元素索引有效时，用下标操作 `[]` 进行数据访问；而当元素索引可能造成越界时，应使用 `at()`。

14.4.1 旁白：设计上的考虑

到目前为止，一切顺利，但为什么我们不在 `operator[](())` 中实现范围检查呢？与我们的

实现类似，标准库 `vector` 在 `operator[]()` 中没有进行范围检查，而是在 `at()` 中提供了范围检查。在本节中，我们将解释这样做的意义所在。主要有以下四个方面因素：

- ❖ 1. 兼容性：在 C++ 具有异常机制之前，人们就已经在使用不带范围检查的下标操作了。
- 2. 效率：你可以在一个不进行范围检查但性能更优的运算符基础上实现一个进行范围检查的运算符，但你不能在一个进行范围检查的运算符基础上实现性能更优的运算符。
- 3. 约束：在一些环境中，异常是不可接受的。
- 4. 检查的可选性：C++ 标准并没有规定你不能对 `vector` 进行范围检查，所以如果你希望进行检查，可选择能够进行范围检查的实现。

14.4.1.1 兼容性

人们总是希望他们以前的代码能够正常运行。例如，如果你编写了一百万行的代码，为在这些代码中正确使用异常而重写代码是一个浩大的工程。我们可能会认为完善这些代码是有意义的，但我们并不是要为此付出时间和金钱的人。而且，已有代码的维护人员常常认为没有进行范围检查的代码原则上是不安全的，但这些特定的代码已经经过了测试，并已使用了很多年，所有错误都已经查找出来了。我们可以对此表示怀疑，但再次强调，我们不是对实际代码做出这种决策并为之负责的人，不应如此武断。在标准库 `vector` 引入 C++ 标准之前，自然不可能有代码使用它，但有数百万行代码都使用了很相似的但不带异常处理的 `vector`（准标准），这些代码中的大部分最终都修改为使用标准 `vector`。

14.4.1.2 效率

在极端情况下，范围检查是一种负担，例如网络接口的缓冲区和高性能科学计算中的矩阵。但是，在我们大多数人大多数时间进行的“普通计算”中，范围检查的代价很少值得关注。因此，我们建议应该尽量地使用进行范围检查的 `vector` 实现。

14.4.1.3 约束

这一论据同样是对某些程序员和程序成立。实际上，它对很多程序员都成立，因此不应轻易忽略。但是，如果你在一个不涉及硬实时要求（参见 25.2.1 节）的环境中开始编写新程序，那么你应该选择基于异常处理及范围检查的 `vector`。

14.4.1.4 检查的可选性

ISO C++ 标准仅仅指出，越界 `vector` 访问不保证有任何特定的语义，且应尽量避免这类访问。当程序试图进行越界访问时，抛出异常是很好地遵循 C++ 标准的处理方式。因此，对特定应用，如果你希望 `vector` 能够抛出异常，并且不关心前述三个论据，那么就应使用进行范围检查的 `vector` 实现。这正是我们在本书中所做的。

概括起来说就是，现实中的程序设计要比我们所希望的更加复杂混乱，但总会存在解决问题的办法。

14.4.2 坦白：使用宏

与我们的 `vector` 类似，大多数标准库 `vector` 实现不对下标运算符（`[]`）进行范围检查，但在 `at()` 中提供范围检查。那么我们程序中的 `std::out_of_range` 异常是从何而来呢？本质上，我们选择了 14.4.1 节中的“选项 4”：`vector` 的实现不必对 `[]` 进行范围检查，但这么做也是允许的，因此我们的代码处理了这种情况。你使用的是我们的调试版本 `Vector`，它对 `[]` 进行了范围检查。这是我们开发代码过程中采用的版本。虽然这一版本会牺牲小部分程序性能，但它有助于减少程序错误和调试时间：

```

struct Range_error : out_of_range { // 增强的 vector 越界错误报告
    int index;
    Range_error(int i) : out_of_range("Range error"), index(i) {}
};

template<typename T> struct Vector : public std::vector<T> {
    using size_type = typename std::vector<T>::size_type;
    using vector<T>::vector; // 使用 vector<T> 的构造函数 (见 15.5 节)

    T& operator[](size_type i) // 不是 return at(i);
    {
        if (i<0||this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }

    const T& operator[](size_type i) const
    {
        if (i<0||this->size()<=i) throw Range_error(i);
        return std::vector<T>::operator[](i);
    }
};

```

通过使用 `Range_error`, 我们能够对元素的越界索引进行调试。由于派生自 `std::vector`, 因此 `Vector` 获得了 `vector` 的所有成员函数。第一个 `using` 为 `std::vector` 的 `size_type` 引入了一个便利的别名; 参见 15.5 节。第二个 `using` 将 `vector` 的所有构造函数都引入了 `Vector`。

在调试复杂程序时这个 `Vector` 版本是十分有用的。另一种替代方法是使用带系统的范围检查的完整标准库 `vector` 实现——实际上, 这可能就是你之前所做的; 我们不可能准确地知道你的编译器和库提供了什么程度的范围检查 (可能会超出 C++ 标准的要求)。

在 `std_lib_facilities.h` 中, 我们采用了一种糟糕的花招 (宏替换), 重新定义 `vector` 使之 ④ 代表 `Vector`:

```
// 令人讨厌的宏替换花招, 来实现带范围检查的 vector:
#define vector Vector
```

这意味着每当你写下 `vector` 时, 编译器看到的都会是 `Vector`。这种花招很糟糕, 因为你所看到的代码与编译器所见的代码并不相同。在实际代码中, 宏是晦涩错误的一个重要来源 (参见 27.8 节和附录 A.17)。

我们也使用了同样方法为 `string` 提供了带范围检查的访问。

遗憾的是, 并不存在标准的、可移植的简洁方法令 `vector` 的口的实现具备范围检查功能。但是, 与我们已采用的方法相比, 用更简洁、完整的方法为 `vector` (和 `string`) 提供范围检查还是可能的。然而, 这通常涉及更换标准库实现, 调整库安装选项, 或者改动标准库的源代码。这些方法在初学者刚开始编程时都不适合——而我们在第 2 章中就已使用了 `string`。

14.5 资源和异常

`vector` 能够抛出异常, 并且我们建议, 当一个函数不能按要求执行操作时, 它应该以抛出异常的方式向其调用者进行报告 (第 5 章)。现在, 是时候介绍如何处理由 `vector` 操作或者我们调用的其他函数抛出的异常了。一种幼稚的回答是——“使用 `try` 语句块捕获异常, 输出一条出错消息并结束程序的运行”——这一方法对于大多数系统而言过于简单了。

编程的一个基本原则是, 如果我们获取了资源, 那么我们还必须负责——直接或间接 圈 地——将其归还给负责管理这些资源的系统。资源的例子包括:

- 内存；
- 锁；
- 句柄；
- 线程句柄；
- 套接字；
- 窗口。

 本质上，资源可以被视为这样一类东西：资源的使用者必须向系统中的“资源管理者”归还（释放）资源，并由“资源管理者”负责资源的回收。最简单的例子就是自由存储区的内存空间，我们通过 new 获得内存空间，而通过 delete 归还内存空间。例如：

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // 获取内存
    ...
    delete[] p; // 释放内存
}
```

如在 12.4.6 节中所学，我们不得不时刻提醒自己释放内存，但这通常不是那么容易的一件事情。当我们学习异常处理时，资源泄漏问题变得更为普遍。特别地，我们需要小心处理那些显式使用 new 操作并将所得指针赋给局部变量的代码，如 suspicious()。

 对于 vector 这样负责释放一个资源的对象，我们称之为资源的所有者（owner）或句柄（handle）。

14.5.1 潜在的资源管理问题

 我们必须小心处理表面上无害的指针赋值操作，如

```
int* p = new int[s]; // 获取内存
```

原因是，在代码中保证每一个 new 操作都对应一个 delete 操作实际上是很困难的。至少在 suspicious() 函数中，必须存在 delete[] p 这样的语句；这样的语句可能会释放内存资源，但也会存在某些意外使得内存的释放不会发生。我们在 ... 中放入什么代码才能造成内存泄漏呢？我们的例子应该能为你带来一些启示并引起你对此类代码的警惕，也应令你更欣赏那些简单有力的替代程序。

当程序运行到 delete 语句时，p 可能已不再指向我们所分配的内存资源：

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // 获取内存
    ...
    if (x) p = q; // 令 p 指向另一个对象
    ...
    delete[] p; // 释放内存
}
```

上述例子中的 if(x) 使得我们不能够确定 p 的取值是否已经改变。程序也可能永远都不能到达 delete 语句：

```
void suspicious(int s, int x)
{
    int* p = new int[s]; // 获取内存
    ...
}
```

```

if (x) return;
//...
delete[] p;           // 释放内存
}

```

程序不能到达 `delete` 语句的原因也许是程序抛出了一个异常：

```

void suspicious(int s, int x)
{
    int* p = new int[s];      // 获取内存
    vector<int> v;
    //...
    if (x) p[x] = v.at(x);
    //...
    delete[] p;           // 释放内存
}

```

我们最关心最后一种情况。当程序员初次遇见这一问题时，他很可能认为这是一个异常处理问题而不是一个资源管理问题。当得出这一错误的判断后，程序员很可能会通过实现异常捕获以试图解决这一问题：

```

void suspicious(int s, int x)    // 混乱的代码
{
    int* p = new int[s];      // 获取内存
    vector<int> v;
    //...
    try {
        if (x) p[x] = v.at(x);
        //...
    } catch (...) {           // 捕获所有异常
        delete[] p;           // 释放内存
        throw;                // 重抛出异常
    }
    //...
    delete[] p;           // 释放内存
}

```

上述解决方法会带来一些额外的代码并造成资源释放代码的重复 (`delete [] p;`)。换句话说，这一解决方法有些丑；更糟的是，它不能很好地推广。考虑下面获取更多资源的例子：

```

void suspicious(vector<int>& v, int s)
{
    int* p = new int[s];
    vector<int> v1;
    //...
    int* q = new int[s];
    vector<double> v2;
    //...
    delete[] p;
    delete[] q;
}

```

注意，如果 `new` 操作不能够分配所需内存，它将抛出标准库异常 `bad_alloc`。对于这个例子，`try...catch` 技术也可以用于解决内存泄漏问题，但在代码中会包含多个 `try` 语句块，这将造成代码的重复冗余。我们不喜欢重复丑陋的代码，因为“重复”意味着代码的维护代价的增加，而“丑陋”意味着代码难于修改、难于阅读，这同样增加了维护代码的代价。



试一试

在上面的例子中添加 `try` 语句块，以保证在产生异常的所有可能情况下，资源都能被正确地释放。

14.5.2 资源获取即初始化

幸运的是，我们可以不必在代码中添加复杂的 `try...catch` 语句就能有效处理潜在的资源泄漏问题。例如：

```
void f(vector<int>& v, int s)
{
    vector<int> p(s);
    vector<int> q(s);
    // ...
}
```

这一实现就好多了。更重要的是，它显然好得多。资源（在这里是自由存储区中的内存空间）由构造函数获取，而由对应的析构函数释放。当解决了向量的内存泄漏问题之后，我们实际上已经解决了这类特别的“异常问题”。这一解决方法具有一般性；它能用于所有资源类型：通过对对象的构造函数获取资源，并通过对应的析构函数释放资源。通过这一方法能够有效处理的资源包括：数据库锁、套接字和 I/O 缓冲区。这一技术有一个拗口的名字“Resource Acquisition Is Initialization”——资源获取即初始化，简写为 RAI^I。

再回到上面的例子。不论我们采用哪种方式退出函数 `f()`，`p` 和 `q` 的析构函数都将被正常调用：因为 `p` 和 `q` 不是指针，我们不能对它们赋值，`return` 语句和异常的抛出均不会妨碍析构函数的执行。当程序的执行序列超出了被完全构造的对象或子对象的作用域时，这些对象的析构函数将自动被调用。对一个对象而言，当其构造函数执行完毕时，它被认为构造完成。探寻这两句话的详细含义是一个让人头疼的事情，但它们的基本含义是对象的构造函数和析构函数会根据实际需要被调用。

特别地，当我们需要在某个作用域内使用可变大小的存储空间时，我们应使用 `vector` 而不是显式使用 `new` 和 `delete`。

14.5.3 保证

当不能只在单一的作用域（及其子作用域）内使用 `vector` 对象时，我们应该怎么做呢？例如：

```
vector<int>* make_vec()           // 创建一个填满的 vector
{
    vector<int>* p = new vector<int>; // 我们在自由存储空间分配 vector
    // ...向 vector 填充数据；这可能抛出异常...
    return p;
}
```

这一例子具有普遍意义：我们调用一个函数构造一个复杂的数据结构，并将该结构作为结果返回。问题是，如果在“填充”`vector` 对象时发生了异常，那么 `make_vec()` 将会造成 `vector` 对象所占内存空间的泄漏。一个不相关的问题是，如果该函数成功了，那么我们不得不通过 `delete` 销毁由 `make_vec()` 返回的对象（参见 12.4.6 节）。

我们可以通过 try 语句块处理异常的抛出：

```
vector<int>* make_vec()           // 创建一个填满的 vector
{
    vector<int>* p = new vector<int>; // 我们在自由存储空间分配 vector
    try {
        // 向 vector 填充数据；这可能抛出异常
        return p;
    }
    catch (...) {
        delete p;      // 进行局部清理
        throw;         // 重抛出异常，允许调用者处理这种情况：
                        // make_vec() 无法完成要求它的工作
    }
}
```

make_vec() 函数展示了错误处理的一个十分通用的形式：函数总是试图完成它的工作，而如果它不能完成工作，则它应释放所有的局部资源（在这里是自由存储区中分配的 vector 对象）并通过抛出异常的方式报告其工作的失败。在这里，异常是由一些其他的函数产生并抛出的（如 `vector::at()`）；`make_vec()` 只是通过 `throw` 直接将该异常重新抛出；这是一种简单而有效地处理错误的方法，并且能够被系统地使用：

- **基本保证：**代码 `try...catch` 的目的是保证 `make_vec()` 要么成功，要么在不造成资源泄漏的前提下抛出异常。这通常称为基本保证。如果程序中的某段代码需要能够从异常 `throw` 中恢复，那么该段代码就需要提供基本保证。所有的标准库代码均提供了基本保证。
- **强保证：**如果一个函数除了提供基本保证，还具有如下特征——在该函数的任务失败后，所有可观测值（所有不属于该函数的值）仍能与其在该函数被调用前的值一致，那么我们称该函数提供强保证。强保证是一种理想情况：函数要么成功完成了所有的任务，要么除了抛出异常之外什么也不做。
- **无抛出保证：**除非我们进行的操作十分简单以至该操作不会产生任何失败和抛出异常，否则我们很可能不能实现同时满足基本保证和强保证的代码。幸运的是，C++ 提供的所有内建工具本质上能够提供无抛出保证：它们不会抛出异常。为了避免异常的抛出，我们应该避免使用 `throw`、`new` 以及引用类型的 `dynamic_cast`（见附录 A.5.7）。

基本保证和强保证对于检验程序的正确性是十分有用的。为了能够根据这些理想情况编写高性能的代码，RAII 是必不可少的。

自然，我们应该一直避免执行未定义的操作（通常它们是有害的），例如对 0 进行解引用，以 0 为除数，以及对数组越界访问。捕获异常并不能保证你不违反这些基本的语言规则。

14.5.4 unique_ptr

因此，`make_vec()` 是一种很有用的函数，在出现异常的情况下，它遵循了好的资源管理的基本原则。在我们想从异常中恢复时，它提供了基本保证——与所有实现良好的函数一样。它甚至能提供强保证，除非在“向 `vector` 填充数据”这部分代码中对非局部数据进行了糟糕操作。尽管如此，`try...catch` 这部分代码仍然是丑陋的。解决方法很明显：我们必须以

某种方式使用 RAII；也就是说，我们需要提供一个对象以容纳 `vector<int>` 对象，以使得当异常发生时它能够销毁 `vector` 对象。为此，在 `<memory>` 中标准库提供了 `unique_ptr`：

```
vector<int>* make_vec()           // 创建一个填满的 vector
{
    unique_ptr<vector<int>> p {new vector<int>}; // 在自由存储空间分配
    // ...向 vector 填充数据；这可能抛出异常...
    return p.release();           // 返回 p 所持有的指针
}
```

`unique_ptr` 是一种能存储指针的对象。我们用 `new` 返回的对象立即初始化 `unique_ptr` 对象。与指针一样，我们可以对 `unique_ptr` 使用 `->` 和 `*` 运算符（例如 `p->at(2)` 或 `(*p).at(2)`），因此我们可以将 `unique_ptr` 看作一种指针。但是，`unique_ptr` 拥有所指向的对象：当销毁 `unique_ptr` 时，它会 `delete` 所指向的对象。这意味着如果在向 `vector<int>` 填充数据时抛出了异常，或者我们过早从返回 `make_vec`，`vector<int>` 会被恰当地销毁。`p.release()` 会从 `p` 中提取出所保存的（指向 `vector<int>` 的）指针，从而我们可以返回它，它还使得 `p` 保存的指针变为 `nullptr`，从而销毁 `p` (`return` 所做的事情) 不会销毁任何对象。

`unique_ptr` 的使用极大地简化了 `make_vec()`。基本上，它令 `make_vec()` 变得与朴素的不安全版本一样简单。重要的是，使用 `unique_ptr` 令我们可以反复做我们建议的事情——用怀疑的目光看待显式的 `try` 块；就像 `make_vec()` 中那样，大多数 `try` 块可以被“资源获取即初始化”技术的某种变体所替代。

使用 `unique_ptr` 的 `make_vec()` 版本已经很好的，唯一问题是它还返回一个指针，从而调用者还是必须记得 `delete` 这个指针。返回一个 `unique_ptr` 可以解决此问题：

```
unique_ptr<vector<int>> make_vec()           // 创建一个填满的 vector
{
    unique_ptr<vector<int>> p {new vector<int>}; // 在自由存储空间分配
    // ...向 vector 填充数据；这可能抛出异常...
    return p;
}
```

`unique_ptr` 非常像普通指针，但它有一项重要限制：你不能将一个 `unique_ptr` 赋予另一个 `unique_ptr` 从而让它们指向相同的对象。此限制是必需的，否则会引起混淆：哪个 `unique_ptr` 拥有所指向的对象？谁负责 `delete` 它？例如：

```
void no_good()
{
    unique_ptr<X> p { new X };
    unique_ptr<X> q {p}; // 错误：幸运的
    ...
} // 此时 p 和 q 都 delete X
```

如果你需要一种既确保释放内存又能被拷贝的“智能”指针，可以使用 `shared_ptr`（见附录 C.6.5）。但是，那是一种更重量级的解决方案，需要一个使用计数来确保最后一个拷贝销毁时能销毁指向的对象。

`unique_ptr` 有一个有趣的性质：与普通指针相比没有额外开销。

14.5.5 以移动方式返回结果

有一种常用的返回大量信息的技术：将信息放在自由存储空间中，然后返回指向它的指针。但这种技术也是高复杂性的来源以及内存管理错误的主要来源：对于从函数返回的指向

自由存储空间的指针，谁 `delete` 它？当发生异常时，我们能否确保指向自由空间中对象的指针被正确 `delete`？除非我们采用了系统的指针管理（或使用 `unique_ptr` 和 `shared_ptr` 这样的“智能”指针），否则答案可能是“好的，我认为是这样的”。而这并不足够好。

幸运的是，当我们向 `vector` 添加移动操作时，就解决了 `vector` 的上述问题：使用移动构造函数将元素的所有权从函数移出。例如：

```
vector<int> make_vec() // 创建一个填满的 vector
{
    vector<int> res;
    // ...向 vector 填充数据；这可能抛出异常...
    return res;           // 移动构造函数高效地转移了所有权
}
```

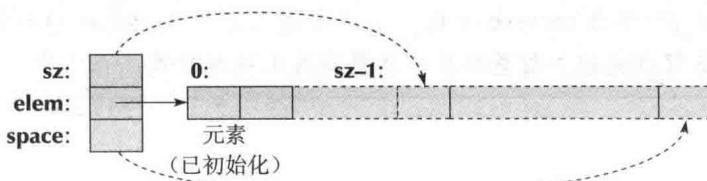
`make_vec()` 的这个（最终）版本最为简单，也是我推荐的版本。移动方法可推广到所有容器以及所有其他资源句柄。例如，`fstream` 使用这种技术跟踪文件句柄。移动方法既简单又通用。使用资源句柄简化了代码并消除了主要错误来源。与直接使用指针的方案相比，没有任何运行时开销，即使有的话，也非常小且容易预测。

14.5.6 `vector` 类的 RAII

使用像 `unique_ptr` 这样的智能指针看上去有点特别。如何保证我们已经发现了所有需要保护的指针？如何保证我们已经释放了所有指向不应在作用域末尾销毁的对象的指针？考虑 14.3.5 节中的 `reserve()`：

```
template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc<=space) return;      // 从不减少分配的内存
    T* p = alloc.allocate(newalloc); // 分配新空间
    for (int i=0; i<sz; ++i) alloc.construct(&p[i], elem[i]); // 拷贝
    for (int i=0; i<sz; ++i) alloc.destroy(&elem[i]);        // 销毁
    alloc.deallocate(elem, space);   // 释放旧空间
    elem = p;
    space = newalloc;
}
```

注意，对已有元素的拷贝操作 `alloc.construct(&p[i], elem[i])` 可能会抛出异常。因此，`p` 是我们在 14.5.1 节中所描述问题的一个例子。我们可以采用 `unique_ptr` 解决方案。一个更好的解决方案是，将“`vector` 所用内存”认为是一种资源；也就是说，我们可以定义一个 `vector_base` 类以代表我们一直使用的基本概念。下图中的三个元素定义了 `vector` 的内存使用：



`vector_base` 的代码（为保持完整性而加入了分配器）如下：

```

template<typename T, typename A>
struct vector_base {
    A alloc;           // 分配器
    T* elem;          // 分配的内存的起始地址
    int sz;            // 元素数目
    int space;         // 分配的空间大小

    vector_base(const A& a, int n)
        : alloc{a}, elem{alloc.allocate(n)}, sz{n}, space{n}{ }
    ~vector_base() { alloc.deallocate(elem, space); }

};


```

注意，`vector_base` 处理的是内存而不是（带类型的）对象。我们的 `vector` 实现可以将它用于存储所需元素类型的对象。本质上，`vector` 是 `vector_base` 的一个便捷的接口：

```

template<typename T, typename A = allocator<T>>
class vector : private vector_base<T,A> {
public:
    ...
};


```

我们可以按如下更简单也更正确的方式重新实现 `reserve()`：

```

template<typename T, typename A>
void vector<T,A>::reserve(int newalloc)
{
    if (newalloc <= this->space) return;      // 从不减少分配的内存
    vector_base<T,A> b(this->alloc, newalloc); // 分配新空间
    uninitialized_copy(b.elem, &b.elem[this->sz], this->elem); // 拷贝
    for (int i=0; i < this->sz; ++i)
        this->alloc.destroy(&this->elem[i]); // 释放旧空间
    swap<vector_base<T,A>>(*this, b);       // 交换表示
}


```

我们使用标准库函数 `uninitialized_copy` 来构造 `b` 中元素的副本，因为它能正确处理元素拷贝构造函数中抛出的异常，而且调用一个函数总比编写一个循环简单。当我们退出 `reserve()` 函数时，原有内存空间将被 `vector_base` 的析构函数自动释放——如果拷贝操作成功的话。如果退出是因拷贝操作抛出异常而造成的，新分配的空间将被释放。`swap()` 函数是一个标准库算法（来自 `<algorithm>`），它能交换两个对象的值。我们使用 `swap<vector_base<T,A>>(*this, b)` 而不是更简单的 `swap(*this, b)`，这是因为 `*this` 和 `b` 是两种不同的类型（分别是 `vector` 和 `vector_base`），因此我们必须显式指出想要使用 `swap` 的哪个特例化版本。类似地，当我们从派生类 `vector<T,A>` 的一个成员来引用基类 `vector_base<T,A>` 的成员时，例如 `vector<T,A>::reserve()`，必须显示使用 `this->`。

试一试

使用 `unique_ptr` 修改 `reserve` 函数。记住在返回前调用 `release()` 函数。将这种方法与 `vector_base` 方法相比较，看看哪种方法更容易正确地实现。

简单练习

1. 定义 `template<typename T> struct S{ T val;};`。

2. 添加构造函数，使得能够对 T 初始化。
3. 定义 S<int>、S<char>、S<double>、S<string> 和 S<vector<int>> 类型的变量，并将其初始化。
4. 读取并打印上述变量的值。
5. 添加函数 get()，该函数返回对 val 的引用。
6. 在类之外编写 get() 的定义。
7. 将 val 设为私有成员。
8. 通过 get() 函数完成练习 4 中的任务。
9. 添加函数模板 set() 以能够改变 val。
10. 通过 S<T>::operator=(const T&) 取代 set()。提示：比 14.2.5 节更简单。
11. 编写 get[] 的 const 版本和非 const 版本。
12. 定义函数 template<typename T> read_val(T&v)，该函数能够将 cin 中读取的值写入 v。
13. 通过 read_val() 读取值，存入练习 3 中前四个变量。
14. 加分题：为 vector<T> 定义输入和输出运算符 (>> 和 <<)。两个运算符都使用 {val, val, val, val} 格式。这使得 read_val() 也能处理 S<vector<int>> 变量。
记住在每一步中对代码进行测试。

思考题

1. 为什么我们需要调整 vector 对象的大小？
2. 为什么我们需要使用具有不同元素类型的 vector 对象？
3. 为什么我们不在所有可能的情况下定义一个具有足够大规模的 vector 对象？
4. 我们需要为一个新的 vector 对象分配多少空闲内存空间？
5. 在何时我们必须将 vector 对象包含的元素拷贝至新的内存空间？
6. 在一个 vector 对象构造成功之后，哪些 vector 操作能够改变它的大小？
7. 拷贝结束后，vector 对象的取值如何？
8. 哪两个操作定义了 vector 的拷贝？
9. 对于类的对象而言，拷贝的默认含义是什么？
10. 什么是模板？
11. 最有用的两种模板参数类型是什么？
12. 什么是泛型编程？
13. 泛型编程与面向对象的编程之间有什么区别？
14. array 与 vector 有什么区别？
15. array 与内置数组有什么区别？
16. resize() 和 reserve() 有什么区别？
17. 什么是资源？给出它的定义并举例说明。
18. 什么是资源泄漏？
19. 什么是 RAII，它能解决什么问题？
20. unique_ptr 的用途是什么？

术语

#define	owner (所有者)	specialization (特例化)
at()	push_back()	strong guarantee (强保证)
basic guarantee (基本保证)	RAII (资源获取即初始化)	template (模板)
exception (异常)	resize()	template parameter (模板参数)
guarantees (保证)	resource (资源)	this
handle (句柄)	re-throw (重抛出)	throw;
instantiation (实例化)	self-assignment (自赋值)	unique_ptr
macro (宏)	shared_ptr	

习题

针对每一习题，创建并测试（通过输出）定义的类的一些对象来验证你的设计和实现确实达到了预期要求。在涉及异常的地方，需要认真考虑错误产生的来源。

- 编写一个模板函数 `f()`，该函数能够将一个 `vector<T>` 的元素加到另一个 `vector<T>` 的元素，例如，`f(v1,v2)` 应该对 `v1` 的每个元素执行 `v1[i]+=v2[i]`。
- 编写一个模板函数，该函数以 `vector<T> vt` 和 `vector<U> vu` 为参数并返回所有 `vt[i]*vu[i]` 之和。
- 编写一个模板类 `Pair`，该类能够存储任何类型的值对。使用该类实现一个类似于我们在计算器中所使用的符号表（见 7.8 节）。
- 将 12.9.3 节中的 `Link` 类修改为模板，该模板以数值类型作为模板参数。然后使用 `Link<God>` 重做第 12 章的习题 13。
- 定义 `Int` 类，该类包含一个 `int` 类的成员。定义该类的构造函数、赋值操作和 `+`、`-`、`*`、`/` 运算符。测试该类并根据需要对它进行完善（例如，定义 `<<` 和 `>>` 运算符实现方便的 I/O）。
- 使用 `Number<T>` 类重新完成上面的习题，其中 `T` 可以是任何数值类型。尝试为 `Number` 实现 `%` 运算符并观察对 `Number<double>` 和 `Number<int>` 进行 `%` 运算的结果。
- 用一些 `Number` 执行习题 2 的程序。
- 用基本分配函数 `malloc()` 和 `free()`（见附录 C10.4）实现一个分配器（见 14.3.7 节）。令 14.4 节结束时定义的 `vector` 能对一些简单的测试用例正确运行。提示：在完整的 C++ 参考手册中查阅“定位 `new`”和“析构函数显式调用”。
- 使用分配器（见 14.3.7 节）重新实现 `vector::operator=()`（见 14.2.5 节）。
- 实现一个简单的 `unique_ptr`，仅需支持构造函数、析构函数、`->`、`*` 和 `release()`。特别地，不要实现其赋值操作或拷贝构造函数。
- 设计并实现 `counted_ptr<T>` 类型，它存储一个指向 `T` 类型对象的指针以及一个指向“使用计数”（一个 `int`）的指针，该整型数被所有指向 `T` 对象的计数指针（`counted_ptr`）所共享。对于一个给定的 `T` 对象，“使用计数”的值应等于指向该对象的计数指针的数目。令 `count_ptr` 的构造函数在自由存储区中为 `T` 对象和其“使用计数”分配内存空间。为 `counted_ptr` 的构造函数接受一个实参作为 `T` 元素的初始值。当最后一个指向 `T` 对象的 `counted_ptr` 被销毁时，`counted_ptr` 的析构函数应负责 `delete T` 对象。为 `counted_ptr` 实现

相关操作，以使我们能够像使用指针一样使用 `counted_ptr`。这一习题是“智能指针”的一个例子，这种指针能够保证一个对象直至其最后一个使用者停止使用它时才被销毁。编写测试 `counted_ptr` 的一组用例，将其用作函数实参、容器元素等等。

12. 定义 `File_handle` 类，该类的构造函数以一个字符串（文件名）作为参数，并且该类在构造函数中打开文件，而在析构函数中关闭文件。
13. 编写 `Tracer` 类，该类的构造函数和析构函数均会打印一个字符串，打印字符串以构造函数参数的形式进行传递。通过 `Tracer` 类观察 RAII 管理对象会在代码中的什么位置完成它们的任务（即通过 `Tracer` 观察局部对象、成员对象、全局对象、由 `new` 分配的对象，等等）。然后，为 `Tracer` 类添加拷贝构造函数和拷贝赋值操作，并通过 `Tracer` 对象观察拷贝是在何时进行的。
14. 为第 13 章习题中的“猎杀怪兽”游戏实现 GUI（用户图形界面）接口和图像输出功能。程序从输入框中获得输入信息，并在一个窗口中显示当前玩家已知的山洞部分。
15. 修改上一习题的程序，以使用户能够在他当前所获得的信息和猜测的基础上标识房间，例如“可能有蝙蝠”和“无底陷阱”。
16. 在有些场合中，人们希望一个空的 `vector` 对象所占用的内存空间尽可能地少。例如，我们可能需要大量使用 `vector<vector<vector<int>>` 类型，但大部分元素均是空向量。定义一个 `vector`，使得 `sizeof(vector<int>) == sizeof(int *)`，即 `vector` 只包含一个指针，该指针指向一个由元素、元素数量、`space` 指针组成的结构。

附言

模板和异常是十分强大的语言特性。它们为编程技术带来了相当的灵活性——主要是允许人们分离关注点，即，一次只处理一个问题。例如，通过使用模板，我们可以在不关注元素类型的情况下设计一个容器，例如 `vector`。类似地，通过使用异常，我们能够将用于检测和报告错误的代码和处理该错误的代码相分离。本章的第三个主题——改变 `vector` 的大小，也体现了这一灵活性：`push_back()`、`resize()` 和 `reserve()` 使我们能够将 `vector` 的定义与 `vector` 的大小说明相分离。

附录 A |

Programming: Principles and Practice Using C++, Second Edition

C++ 语言概要

慎重许愿，它有可能成真。

——俗语

本附录概述 C++ 语言的一些重要特性。本附录的内容都是精心选择的，特别适合于那些希望接触一些超出本书主题之外内容的初学者。本附录的目标是简洁扼要，而非完整性。

A.1 一般内容

本附录的目的是作为补充参考资料，而不是像其他章节一样需要从头到尾仔细阅读。它（或多或少地）系统描述了 C++ 语言的重要特性。本附录不是完整的参考文献，而只是概述。重点内容都是根据教学过程中学生提出的问题确定的。通常，你需要查看相关章节来获得更为详细完整的解释。本附录不追求与 C++ 标准相同的精确性和术语，而是追求易于查阅。更详细的信息可参考 Stroustrup 的《The C++ Programming Language》一书。ISO C++ 标准定义了 C++ 语言，但其文档并不是为了初学者所编写的，并不适合入门阅读学习。不要忘记使用在线文档。如果你是在学习本书较早章节时查阅本附录，要有心理准备，一些内容看起来很“神秘”，不必担心，这些内容应该是在稍后章节中详细介绍的。

标准库的相关内容在附录 C 中介绍。

C++ 标准由 ISO（国际标准组织）下属的一个委员会负责制订，标准制订过程中也与一些国家的标准组织进行了合作，如 INCITS（美国）、BSI（英国）和 AFNOR（法国）。当前的版本是 ISO/IEC 14882:2011 C++ 程序设计语言标准。

A.1.1 术语

C++ 标准定义了什么是 C++ 程序，及其语言特性的含义：

- 符合标准的（conforming）：如果按照标准定义，一个程序被认可是 C++ 程序，则称之为符合标准的（或者通俗地讲，合法的或有效的）。
- 实现定义的（implementation defined）：程序可以（而且通常的确是）依赖于那些只对给定编译器、操作系统、机器架构等等才有明确定义的语言特性（如 int 占用的内存大小，以及 'a' 的数值等等）。这些由具体实现定义的特性在 C++ 标准中都会列出，而在具体实现的文档中应该明确说明，其中很多特性是在标准头文件，如 `<limits>`（参见附录 C.1.1）中定义的。因此，符合标准的程序未必能移植到所有 C++ 实现之上。
- 未说明的（unspecified）：一些语言特性的含义是未说明的、未定义的或者是不符合标准但无须检测的。显然，最好不要使用这些特性，本书就是这样做的。应该避免的未说明特性包括：
 - 不同源文件中的不一致的定义（应该一致地使用头文件，参见 8.3 节）。

- 在一个表达式中对同一个变量重复读写（最典型的例子 `a[i]=++i;`）。
- 大量使用显式类型转换，特别是 `reinterpret_cast`。

A.1.2 程序开始和结束

一个 C++ 程序必须包含唯一一个名为 `main()` 的全局函数，它是程序执行的起点。`main()` 的返回类型是 `int` (`void` 返回类型并不符合标准)。`main()` 的返回值将作为程序的返回值提交给“系统”。某些系统会忽略这个返回值，但返回 0 通常意味着程序正常结束，返回非 0 值或者抛出一个异常（但异常未被捕获的情况被认为是一种糟糕的风格）表示程序失败。

`main()` 的参数可以由具体实现定义，但所有实现都必须接受两种方式（虽然每个程序只会用到其中某一个）：

```
int main();           // 无参
int main(int argc, char* argv[]); // argv[] 保存 argc 个 C 风格字符串
```

`main()` 不需要显式返回一个值。如果它没有明确返回一个值，而是“一落到底”，则意味着返回 0。下面是最简短的 C++ 程序：

```
int main() {}
```

如果你定义了一个全局（名字空间）作用域的对象，且它具有构造函数和析构函数，那么逻辑上，构造函数会在“`main()` 之前”执行，而析构函数则在“`main()` 之后”执行（从技术角度看，执行构造函数实际上是调用 `main()` 的工作的一部分，而执行析构函数则是从 `main()` 返回工作的一部分）。只要可能，就不要使用全局对象，特别是有较为复杂的构造和析构过程的全局对象。

A.1.3 注释

能通过代码表达清楚的，就应该用代码表达。不过，C++ 提供了两种风格的注释，帮助程序员描述代码表达不好的内容：

```
// 这是一个行注释
```

```
/*
    这是一个
    块注释
*/
```

显然，块式注释风格更多地用于多行注释，不过有些人即使是对多行注释也更喜欢使用单行注释风格：

```
// 这是一个
// 多行注释
// 使用三个单行注释表达
```

```
/* 这是一个单行注释，使用一个块注释表达 */
```

注释是描述代码意图的必要的文档形式，参见 7.6.4 节。

A.2 字面值常量

字面值常量 (literal) 用于表示不同类型的值。例如，12 表示整型值“十二”，“Morning”表示字符串值 Morning，而 `true` 表示布尔值“真”。

A.2.1 整数字面值常量

整数字面值常量有三种形式：

- 十进制：十进制数字序列
十进制数字：0、1、2、3、4、5、6、7、8 和 9
- 八进制：以 0 开始的八进制数字序列
八进制数字：0、1、2、3、4、5、6 和 7
- 十六进制：以 0x 或 0X 开始的十六进制数字序列
十六进制数字：0、1、2、3、4、5、6、7、8、9、a、b、c、d、e、f、A、B、C、D、E 和 F
- 二进制：以 0b 或 0B 开始的二进制数字序列（C++14 新特性）
二进制数字：0、1

后缀 u 或 U 表示无符号整数（见 25.5.3 节），而后缀 l 或 L 表示长整型，例如 10u 和 123456UL。

C++14 还允许在数值字面值常量使用单引号作为数字分隔符。例如，0b0000'0001'0010'0011 表示 0b0000000100100011，而 1'000'000 表示 1000000。

A.2.1.1 数制系统

我们通常书写数值使用的都是十进制表示法。123 表示一百加二十加三，即 $1*100+2*10+3*1$ ，或者 $1*10^2+2*10^1+3*10^0$ （ \wedge 表示“幂”）。十进制（decimal）还被称作以 10 为基底（base-10）。10 在这里实际上没有任何特殊之处，我们想表达的是 $1*base^2+2*base^1+3*base^0$ ，只不过本例中 `base==10` 而已。有很多理论试图解释人类为什么使用基底 10，其中一种理论已经“扎根于”一些自然语言中了：人类有十根手指；在一个按位的数值系统中，每个直接表示数值的符号（如 0、1、2）被称为数字（digit），在拉丁语中“digit”意为手指（英语的 finger）。

我们偶尔也使用其他数制。典型的例子包括：计算机内存中的正整数用二进制表示（用材料的不同物理状态稳定表示 0 和 1 相对容易些）；人们处理低层硬件问题时有时使用八进制；内存中的内容更多是用十六进制访问。

我们来看一下十六进制。我们需要为 0 到 15 这 16 个数字命名，通常使用 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F 这 16 个符号，其中 A 表示十进制值 10，B 表示 11，依此类推：

A==10, B==11, C==12, D==13, E==14, F==15

我们现在就可以将十进制值 123 写作十六进制值 7B 了。为了理解两者是相等的，我们考察十六进制值 7B 所表示的含义 $7*16+11$ ，恰好等于（十进制的）123。反过来，十六进制值 123 表示 $1*16^2+2*16+3$ ，即 $1*256+2*16+3$ ，其值等于（十进制的）291。如果你从未接触过非十进制的整数表示方式，我们强烈建议你尝试将一些整数从十进制转换为十六进制。注意，十六进制数字与二进制值之间存在一种简短的对应关系：

十六进制和二进制								
十六进制	0	1	2	3	4	5	6	7
二进制	0000	0001	0010	0011	0100	0101	0110	0111

(续)

十六进制和二进制								
十六进制	8	9	A	B	C	D	E	F
二进制	1000	1001	1010	1011	1100	1101	1110	1111

十六进制和二进制之间的关系，可以解释十六进制表示为什么那么流行。特别是，一个字节的值可以简单地用两个十六进制数字表示。

在 C++ 中，(幸运的是) 默认情况下数值是用十进制表示的。为了表示十六进制数，需要使用前缀 `0X` (`X` 表示 “hex”)，因此 `123==0X7B`, `0X123==291`。使用小写的 `x` 是完全等价的表示方式，因此，`123==0x7B` 和 `0x123==291` 也是正确的。类似地，十六进制数字也可以用小写字面 `a`、`b`、`c`、`d`、`e` 和 `f` 来表示，例如 `123=0x7b`。

八进制是以 8 为基底的。我们只需八个数字：0、1、2、3、4、5、6、7。在 C++ 中，八进制数以 0 开头，因此 0123 不是十进制数 123，而是 $1*8^2+2*8+3$ ，即 $1*64+2*8+3$ ，等于 (十进制) 83。反过来，八进制数 83，即 083，表示 $8*8+3$ ，等于 (十进制) 67。用 C++ 语法表示，就是 `0123==83`, `083==67`。

二进制是以 2 为基底的。我们只需两个数字：0 和 1。在 C++ 中，我们无法用字面值常量直接表示二进制值。C++ 的字面值常量和输入输出只直接支持八进制、十进制和十六进制。不过，即使我们无法用 C++ 代码直接表示二进制数，了解二进制的基本知识还是很有用的。例如，(十进制) 123 可以表示为：

`1*2^6+1*2^5+1*2^4+1*2^3+0*2^2+1*2+1`

也就是 $1*64+1*32+1*16 + 1*8+0*4+1*2+1$ ，用二进制表示就是 1111011。

A.2.2 浮点字面值常量

一个浮点字面值常量包含一个小数点 (.)，一个指数 (如 `e3`)，或者通过后缀 `d` 或 `f` 指明是浮点值。例如：

```
123      // int (无小数点、后缀或指数)
123.     // double: 123.0
123.0    // double
123     // double: 0.123
0.123   // double
1.23e3   // double: 1230.0
1.23e-3  // double: 0.00123
1.23e+3  // double: 1230.0
```

如果没有通过后缀特别指明，浮点字面值常量的类型为 `double`，例如：

```
1.23     // double
1.23f    // float
1.23L    // long double
```

A.2.3 布尔字面值常量

`bool` 类型的字面值常量有 `true` 和 `false`，前者的整型值为 1，而后的整型值为 0。

A.2.4 字符字面值常量

字符字面值常量是用单引号包含起来的字符，如 `'a'` 和 `'@'`。另外，还有一些“特殊字符”：

名 称	ASCII 名	C++ 名
换行 (newline)	NL	\n
水平制表符 (horizontal tab)	HT	\t
垂直制表符 (vertical tab)	VT	\v
退格 (backspace)	BS	\b
回车 (carriage return)	CR	\r
换页 (form feed)	FF	\f
警告 (alert)	BEL	\a
反斜线 (backslash)	\	\\\
问号 (question mark)	?	\?
单引号 (single quote)	'	'
双引号 (double quote)	"	\"
八进制数 (octal number)	ooo	\ooo
十六进制数 (hexadecimal number)	hhh	\xhhh

特殊字符通过用单引号包含“C++ 名”来表示，例如 '\n' (换行) 和 '\t' (制表符)。

字符集还包括如下可见字符：

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!@#$%^&*()_+|~{}[]:"';<>?,./
```

如果你希望编写可移植的代码，就不应该依赖其他可见字符。字符的整型值，如 a 的值 'a'，是依赖于实现的（但很容易获取，如 cout<<int('a')）。

A.2.5 字符串字面值常量

字符串字面值常量是双引号包含起来的字符序列，例如：“Knuth”和“King Canute”。字符串中不能包含换行符，应该用特殊字符 \n 代替：

```
"King
Canute"      // 错误：字符串字面常量中包含换行
"King\nCanute" // 正确：在字符串字面值常量中包含换行的正确方法
```

两个仅用空白符分隔开的字符串字面值常量会被连接为一个字符串，例如：

```
"King" "Canute" // 等价于 "KingCanute" (无空格)
```

注意，特殊字符（例如 \n）可以出现在字符串字面值常量中。

A.2.6 指针字面值常量

指针字面值常量只有一个：空指针 nullptr。便利起见，任何值为 0 的常量表达式都可以当作空指针使用，例如：

```
t* p1 = 0;      // 正确：空指针
int* p2 = 2-2;  // 正确：空指针
int* p3 = 1;    // 错误：1 是一个 int，不是一个指针
int z = 0;
int* p4 = z;   // 错误：z 不是一个常量
```

0 被隐式转换为空指针。

在 C++ 中（但 C 中不是，因此要小心 C 头文件），NULL 被定义为 0，因此可以这样编写代码：

```
int* p4 = NULL; // (如有正确的 NULL 定义) 空指针
```

A.3 标识符

标识符（identifier）是以字母和下划线开头，后接 0 个或多个（大写或小写）字母、数字或下划线的序列：

```
int foo_bar;      // 正确
int FooBar;       // 正确
int foo bar;      // 错误：标识符中不能包含空格
int foo$bar;       // 错误：标识符中不能包含 $
```

以下划线开头或包含连续两个下划线的标识符是为 C++ 实现预留的，你的程序中不要使用这种标识符。例如：

```
int _foo;          // 不要这样用
int foo_bar;       // 正确
int foo__bar;      // 不要这样用
int foo_;          // 正确
```

A.3.1 关键字

关键字（keyword）是语言自己用来表示语言结构的特殊标识符：

关键字（保留的标识符）					
alignas	class	explicit	noexcept	signed	typename
alignof	compl	export	not	sizeof	union
and	concept	extern	not_eq	static	unsigned
and_eq	const	false	nullptr	static_assert	using
asm	const_cast	float	operator	static_cast	virtual
auto	constexpr	for	or	struct	void
bitand	continue	friend	or_eq	switch	volatile
bitor	decltype	goto	private	template	wchar_t
bool	default	if	protected	this	while
break	delete	inline	public	thread_local	xor
case	do	int	register	throw	xor_eq
catch	double	long	reinterpret_cast	true	
char	dynamic_cast	mutable	requires	try	
char16_t	else	namespace	return	typedef	
char32_t	enum	new	short	typeid	

A.4 作用域、内存类别和生命周期

C++ 中所有名字（预处理器名字除外，参见附录 A.17）都存在于一个作用域中，也就是说，名字属于某个代码区域，在此区域中名字才可以使用。数据（对象）都保存于内存中某个位置，用于保存对象的内存类型称为存储类别（storage class）。一个对象的生命周期

(lifetime) 从其初始化的时刻开始, 到它被销毁的时刻为止。

A.4.1 作用域

C++ 中有五种作用域 (见 8.4 节):

- 全局作用域 (global scope): 除非名字定义于某个语言结构中 (如一个类或者一个函数), 否则其作用域为全局作用域。
- 名字空间作用域 (namespace scope): 如果名字定义于一个名字空间中, 而且不在某个语言结构中 (如一个类或者一个函数), 则其作用域为名字空间作用域。从技术角度看, 全局作用域可以看作“名字为空”的名字空间作用域。
- 局部作用域 (local scope): 如果名字在一个函数内声明 (包括函数的参数), 则其作用域为局部作用域。
- 类作用域 (class scope): 如果名字是类的一个成员, 则其作用域为类作用域。
- 语句作用域 (statement scope): 如果名字声明是在一个 `for`、`while`、`switch` 或者 `if` 语句中, 则其作用域是语句作用域。

一个变量的作用域会延伸到定义它的语句的末尾, 例如:

```
for (int i = 0; i < v.size(); ++i) {
    // i 可用于此处
}
if (i < 27)           // for 语句中的 i 的作用域不包括这里
```

对于类作用域和名字空间作用域来说, 由于类有类名, 名字空间本身也有名字, 因此可以在“其他地方”引用它们的成员, 例如:

```
void f();           // 在全局作用域中

namespace N {
    void f()       // 在名字空间作用域 N 中
    {
        int v;     // 在局部作用域中
        ::f();      // 调用全局的 f()
    }
}

void f()
{
    N::f();       // 调用 N 的 f()
}
```

如果你调用 `N::f()` 或是 `::f()`, 会发生什么? 请参考附录 A.15。

A.4.2 内存类别

C++ 中有三种类型的内存空间 (见 12.4 节):

- 自动内存 (automatic storage): 除非显式声明为 `static`, 否则在函数中定义的变量 (包括函数参数) 存储在自动内存空间中 (即“栈”中)。当函数被调用时, 会为其分配自动内存空间, 当它返回时, 这部分空间被释放。因此, 如果函数 (直接地或间接地) 调用自身, 其自动变量 (参数) 会存在多个副本, 每个副本对应一次调用 (见 8.5.8 节)。

- 静态内存 (static storage): 全局变量和名字空间中声明的变量存储在静态内存空间中，在函数和类中声明的变量，如果显式声明为 static，也存储在静态内存中。连接器在“程序开始运行之前”分配静态内存空间。
- 自由内存 (free storage, 又叫堆 (heap)): 用 new 创建的对象保存在自由内存空间中：例如：

```

vector<int> vg(10);      // 在程序开始时 (“main() 之前”) 构造一次

vector<int>* f(int x)
{
    static vector<int> vs(x); // 仅在第一次调用 f() 之前构造
    vector<int> vf(x+x);    // 每次调用 f() 时构造

    for (int i=1; i<10; ++i) {
        vector<int> v1(i); // 每个迭代步中构造
        // ...
    }                      // v1 在此处 (每个迭代步结束) 被销毁

    return new vector<int>(vf); // 在自由内存空间中构造, 成为 vf 的副本
}                          // vf 在此处被销毁

void ff()
{
    vector<int>* p = f(10); // 从 f() 获取 vector
    // ...
    delete p;           // 释放从 f() 得到的 vector
}

```

其中，静态分配的变量 vg 和 vs 在程序结束时 (“main() 之后”) 被销毁，当然前提是它们在程序开始前被创建。

类的成员不是这样分配的。当你在某处分配一个类对象时，其非静态成员也保存在此处（与对象位于相同的内存空间中）。

代码与数据是分开存放的。例如，并不是每个类对象都保存着成员函数的代码，每个成员函数只保存一份，与其他程序代码一起存储在某个地方。

参见 19.3 节和 12.7 节。

A.4.3 生命期

在使用之前，对象必须进行初始化。初始化可以通过显式地初始化代码来完成，也可以隐式地通过构造函数或者内置类型的默认初始化规则来完成。对象生命期的终点由其作用域和所处内存空间类别决定（见 12.4 节和附录 C.4.2）：

- 局部（自动）对象 (local object, automatic object): 在程序执行到它定义的位置时被创建，在程序执行到它的作用域末尾时被销毁。
- 临时对象 (temporary object): 由特定的子表达式创建，在其完整表达式求值完毕后销毁。“完整表达式”指不包含在任何其他表达式的表达式（非子表达式）。
- 名字空间对象和静态类成员 (namespace object, static class member) 在程序开始时 (“main() 之前”) 被创建，在程序结束时 (“main() 之后”) 被销毁。
- 局部静态对象 (local static object): 在程序执行到它定义的位置时被创建，在程序结束时销毁。

- 自由内存空间对象 (free-store object): 由 `new` 创建, 可以用 `delete` 来销毁。与局部或名字空间引用绑定的临时变量, 其生命周期与引用的生命期相同。例如:

```

const char* string_tbl[] = { "Mozart", "Grieg", "Haydn", "Chopin" };
const char* f(int i) { return string_tbl[i]; }
void g(string s){}

void h()
{
    const string& r = f(0);      // 将临时 string 绑定到 r
    g(f(1));                   // 创建一个临时 string 并作为参数传递
    string s = f(2);           // 用临时 string 初始化 s
    cout << "f(3): " << f(3) // 创建一个临时 string 并作为参数传递
    << " s: " << s
    << " r: " << r << '\n';
}

```

运行结果如下:

f(3): Chopin s: Haydn r: Mozart

调用 `f(1)`、`f(2)` 和 `f(3)` 所创建的临时 `string` 在其所在表达式求值完毕后就被销毁了。但为调用 `f(0)` 所创建的临时变量与 `r` 绑定了, 因此它会“存活”至 `h()` 结束。

A.5 表达式

本节对 C++ 的运算符进行概述。我们使用一些助记符缩写, 如 `m` 表示成员名, `T` 表示类型名, `p` 表示结果为指针的表达式, `x` 表示表达式, `v` 表示左值表达式, `lst` 表示参数列表。算术运算的结果类型由“常用算术类型转换规则”决定 (见附录 A.5.2.2)。本节的内容都是针对内置运算符的, 并不讨论用户自定义运算符。当然, 当你自定义运算符时, 我们鼓励你遵循内置运算符的语义规则 (见 9.6 节)。

作用域解析运算符

<code>N::m</code>	<code>m</code> 在名字空间 <code>N</code> 之中, <code>N</code> 为一个名字空间或者一个类
<code>::m</code>	<code>m</code> 在全局名字空间之中

注意, 成员是可以嵌套的, 所以 `N::C::m` 是合法的, 参见 8.7 节。

后缀运算

<code>x.m</code>	成员访问, <code>x</code> 必须是一个类对象
<code>p->m</code>	成员访问, 等价于 <code>(*p).m</code> , <code>p</code> 必须是一个类对象指针
<code>p[x]</code>	下标, 等价于 <code>*(p+x)</code>
<code>f(lst)</code>	函数调用, 调用 <code>f</code> , 参数列表为 <code>lst</code>
<code>T((lst))</code>	构造对象: 构造一个类型为 <code>T</code> 的对象, 参数列表为 <code>lst</code>
<code>v++</code>	(后) 增 1 运算, 表达式 <code>v++</code> 的值为 <code>v</code> 增 1 之前的值
<code>v--</code>	(后) 减 1 运算, 表达式 <code>v--</code> 的值为 <code>v</code> 减 1 之前的值
<code>typeid(x)</code>	运行时类型识别 (对表达式 <code>x</code>)
<code>typeid(T)</code>	运行时类型识别 (对类型 <code>T</code>)
<code>dynamic_cast<T>(x)</code>	带类型检查的运行时类型转换, 将 <code>x</code> 转换为类型 <code>T</code>
<code>static_cast<T>(x)</code>	带类型检查的静态 (编译时) 类型转换, 将 <code>x</code> 转换为类型 <code>T</code>
<code>const_cast<T>(x)</code>	不做检查的类型转换, 从 (向) <code>x</code> 的类型中去掉 (加上) <code>const</code> , 并将其转换为类型 <code>T</code>
<code>reinterpret_cast<T>(x)</code>	不做检查的类型转换, 通过重新解释 <code>x</code> 的位模式将其转换为类型 <code>T</code>

本书并未讨论 `typeid` 运算符及其使用，请查阅专家级的书籍或资料。注意，类型转换操作不会修改实参，而是根据实参值创建一个所需类型的新的对象，参见附录 A.5.7。

单目运算

<code>sizeof(T)</code>	类型 T 的大小 (字节数)
<code>sizeof(x)</code>	表达式 x 的类型的大小 (字节数)
<code>++v</code>	(前) 增 1 运算，与 <code>v+=1</code> 等价
<code>--v</code>	(前) 减 1 运算，与 <code>v-=1</code> 等价
<code>~x</code>	x 的补 (二进制运算)，~ 是位操作
<code>!x</code>	x 的非，返回 true 或 false (逻辑运算)
<code>&v</code>	v 的地址
<code>*p</code>	p 指向的对象内容
<code>new T</code>	在自由内存空间中分配一个类型为 T 的对象
<code>new T(lst)</code>	在自由内存空间中分配一个类型为 T 的对象，并用 lst 对其进行初始化
<code>new (lst) T</code>	在 lst 指定的位置构造一个类型为 T 的对象
<code>new (lst) T(lst2)</code>	在 lst 指定的位置构造一个类型为 T 的对象，并用 lst2 对其进行初始化
<code>delete p</code>	释放 p 指向的对象
<code>delete[] p</code>	释放 p 指向的数组
<code>(T)x</code>	C 风格类型转换，将 x 的类型转换为 T

注意，用 `delete` 释放的对象和数组必须是用 `new` 分配的内存空间，参见附录 A.5.6。注意，`(T)x` 非常不明确，因此比更加明确的其他类型转换运算符更容易出错，参见附录 A.5.7。

成员运算

<code>x.*ptm</code>	成员指针 ptm 所指向的 x 的成员
<code>p->*ptm</code>	成员指针 ptm 所指向的 *p 的成员

本书并未介绍成员指针 (pointer-to-member) 特性，请查阅专家级别的参考书籍。

乘法运算符

<code>x*y</code>	x 乘以 y
<code>x/y</code>	x 除以 y
<code>x%y</code>	x 模 y (x 除以 y 的余数，x 和 y 不能是浮点类型)

如果 `y==0`, `x/y` 和 `x%y` 的结果是未定义的。如果 `x`、`y` 是负数，`x%y` 的结果是由具体实现定义的。

加法运算符

<code>x+y</code>	x 加 y
<code>x-y</code>	x 减 y

移位运算符

<code>x<<y</code>	x 左移 y 位
<code>x>>y</code>	x 右移 y 位

对于 `>>` 和 `<<` 用于（内置类型）二进制移位，请参考 25.5.4 节。如果最左边的运算对象是一个 `iostream`，则这两个运算符表示 I/O 操作，参见第 10 章和第 11 章。

关系运算符

<code>x<y</code>	<code>x</code> 小于 <code>y</code> , 返回一个 <code>bool</code> 值
<code>x<=y</code>	<code>x</code> 小于等于 <code>y</code>
<code>x>y</code>	<code>x</code> 大于 <code>y</code>
<code>x>=y</code>	<code>x</code> 大于等于 <code>y</code>

关系运算符的计算结果是 `bool` 类型。

相等关系运算符

<code>x==y</code>	<code>x</code> 等于 <code>y</code> , 返回一个 <code>bool</code> 值
<code>x!=y</code>	<code>x</code> 不等于 <code>y</code>

注意, `x!=y` 即 `!(x==y)`。相等关系运算符的计算结果也是一个 `bool` 值。

位与运算符

<code>x&y</code>	<code>x</code> 和 <code>y</code> 的位与
----------------------	-------------------------------------

注意, `&` (与 `^`、`|`、`~`、`>>` 和 `<<` 类似) 处理二进制位集合。例如, 如果 `a` 和 `b` 是 `unsigned char` 类型, `a&b` 也是 `unsigned char` 类型, 其每一位是 `a` 和 `b` 的对应位进行 `&` (与) 的结果, 参见附录 A.5.5。

位异或运算符

<code>x^y</code>	<code>x</code> 和 <code>y</code> 的位异或
------------------	--------------------------------------

位或运算符

<code>x y</code>	<code>x</code> 和 <code>y</code> 的位或
------------------	-------------------------------------

逻辑与运算符

<code>x&&y</code>	<code>x</code> 和 <code>y</code> 的逻辑与, 返回 <code>true</code> 或 <code>false</code> , 只有 <code>x</code> 为真时才对 <code>y</code> 求值
---------------------------	---

逻辑或运算符

<code>x y</code>	<code>x</code> 和 <code>y</code> 的逻辑或, 返回 <code>true</code> 或 <code>false</code> , 只有 <code>x</code> 为假时才对 <code>y</code> 求值
-------------------	---

参见附录 A.5.5。

条件运算符

<code>x?y:z</code>	若 <code>x</code> 为真则结果为 <code>y</code> , 否则结果为 <code>z</code>
--------------------	---

例如:

```
template<class T> T& max(T& a, T& b) { return (a>b)?a:b; }
```

“问号冒号运算符”的相关内容参见 8.4 节。

赋值运算符

$v=x$	将 x 的值赋予 v , 表达式的结果为 v 的结果
$v^*=x$	大致等价于 $v=v^*(x)$
$v/=x$	大致等价于 $v=v/(x)$
$v\% =x$	大致等价于 $v=v\%(x)$
$v+=x$	大致等价于 $v=v+(x)$
$v-=x$	大致等价于 $v=v-(x)$
$v>>=x$	大致等价于 $v=v>>(x)$
$v<<=x$	大致等价于 $v=v<<(x)$
$v\&=x$	大致等价于 $v=v\&(x)$
$v\wedge=x$	大致等价于 $v=v\wedge(x)$
$v =x$	大致等价于 $v=v (x)$

这里“大致等价于 $v=v^*(x)$ ”的意思是： $v^*=x$ 与 $v=v^*(x)$ 相同，差别只在于前者只对 v 求值一次。例如， $v[++i]^*=7+3$ 意为 $(++i, v[i]=v[i]^*(7+3))$ ，而不是 $(v[++i]=v[++i]^*(7+3))$ （后者的求值是未定义的，参见 8.6.1. 节）。

抛出操作

<code>throw x</code>	抛出 x 的值
----------------------	-----------

`throw` 表达式的类型为 `void`。

逗号运算符

x,y	先执行 x , 再执行 y , 表达式的结果为 y
-------	---------------------------------

位于同一个表格中的运算符的优先级都是相同的，越靠前的表格中的运算符优先级越高。例如， $a+b*c$ 意味着 $a+(b*c)$ 而不是 $(a+b)*c$ ，因为 $*$ 的优先级比 $+$ 高。类似地， $*p++$ 意味着 $*(p++)$ 而不是 $(*p)++$ 。单目运算符和赋值运算符是右结合的，其他运算符都是左结合的。例如， $a=b=c$ 意味着 $a=(b=c)$ ，而 $a+b+c$ 意味着 $(a+b)+c$ 。

左值 (lvalue) 是表示对象的表达式，该对象理论上是可以被修改的（但显然，如果左值的类型是 `const` 的，则类型系统不允许对其进行修改）、可以获得其地址的。与左值相对应的是右值 (rvalue)，即，右值表达式表示的对象是不可修改的或者不能获得地址的，例如，函数返回值就是右值（因此，`&f(x)` 是错误的）。

A.5.1 用户自定义运算符

本节给出的表达式规则都是针对内置类型的。如果使用的是用户自定义类型，表达式可以简单地转换为对恰当的用户自定义运算符函数的调用，因而其运算结果是由函数调用规则决定的，而非表达式规则。例如：

```
class Mine {/* ... */;
bool operator==(Mine, Mine);

void f(Mine a, Mine b)
```

```

{
    if (a==b) {      // a==b 表示 operator==(a,b)
        ...
    }
}

```

用户自定义类型或者是一个类（见附录 A.12 和第 9 章），或者是一个枚举（见附录 A.11 和 9.5 节）。

A.5.2 隐式类型转换

整型和浮点类型（见附录 A.8）可以在赋值和表达式中随意混合使用。只要可能，编译器就会对值进行（隐式）类型转换，以避免丢失信息。不幸的是，导致信息丢失的类型转换也是隐式进行的。

A.5.2.1 提升

能保持值不变的隐式类型转换通常被称为提升（promotion）。在进行算术运算之前，会对短整型进行整型提升（integral promotion），将其转换为 int 型。这个操作反映了提升的最初目的：将运算对象转换为算术运算中的“自然”大小。另外，将 float 转换为 double 也被认为是一种提升。

提升是常用的算术类型转换的一部分（见附录 A.5.2.2）。

A.5.2.2 类型转换

基本数据类型之间的转换方式非常多，容易令人迷惑。当编写代码时，你必须一直小心避免未定义的行为和类型转换，以免信息无声无息地被丢掉（见 3.9 节和 25.5.3 节）。编译器可以对很多可疑的类型转换给出警告：

- 整型转换（integral conversion）：一个整型值可以转换为其他整数类型。一个枚举值可以转换为整数类型。如果目标类型是无符号的，转换结果是从源中取出尽量多的二进制位放入目标中（如果目标类型无法容纳源的所有二进制位，则将高位丢弃）。如果目标类型是有符号的，而且值可以用目标类型表示的话，则值不会改变；否则，如果目标类型无法容纳值，则最终结果由具体实现定义。注意，bool 和 char 都是整数类型。
- 浮点类型转换（floating-point conversion）：一个浮点值可以转换为其他浮点类型。如果源值可以用目标类型精确描述，则转换结果就是源值。否则，如果源值在两个目标类型值之间，则转换结果取其中之一。否则，结果是未定义的。注意，float 转换为 double 被认为是提升。
- 指针和引用转换（pointer and reference conversion）：任何对象的指针类型都可以隐式转换为 void*（见 12.8 节和 27.3.5 节）。派生类指针（引用）可以隐式转换为可访问的、明确的基类指针（引用）（见 19.3 节）。求值结果为 0 的常量表达式（见附录 A.5 和 4.3.1 节）可以隐式转换为任意指针类型。T* 可以隐式转换为 const T*。类似地，T& 可以隐式转换为 const T&。
- 布尔类型转换（boolean conversion）：指针、整数和浮点数都可以隐式转换为 bool 类型，非 0 值转换为 true，0 转换为 false。
- 浮点数和整数的转换（floating-integral conversion）：当一个浮点值转换为一个整型值时，小数部分被丢弃。换句话说，浮点型到整型的转换是截断转换。如果目标类型无法容纳截断值，结果是未定义的。整型到浮点型的转换在数学上是正确的，也是

硬件允许的。如果整型值无法用浮点型值精确表示，就会丢失精度。

- 常用算术类型转换 (usual arithmetic conversion)：在二元运算中，会对运算对象进行这种转换，将它们转换为相同类型，将其作为结果类型：

1. 如果其中一个运算对象的类型是 `long double`，另一个也被转换为 `long double`。否则，如果其中一个运算对象是 `double` 类型，另一个也被转换为 `double` 类型。否则，如果其中一个运算对象是 `float`，另一个也被转换为 `float`。否则，对两个运算对象都进行整数提升。
2. 接下来，如果其中一个运算对象是 `unsigned long`，另一个也被转换为 `unsigned long`。否则，如果一个运算对象是 `long int`，另一个是 `unsigned int`，那么如果 `long int` 可以表示所有 `unsigned int` 值，则将 `unsigned int` 类型的运算对象转换为 `long int`，否则两个运算对象都被转换为 `unsigned long int`。否则，如果其中一个运算对象是 `long`，另一个也被转换为 `long`。否则，如果其中一个运算对象是 `unsigned`，另一个也被转换为 `unsigned`。否则，两个运算对象都是 `int`。

显然，最好不要在表达式中使用过于复杂的混合类型运算，这样就能尽量减少隐式的类型转换，避免意料之外的错误。

A.5.2.3 用户自定义转换

除了标准的类型提升和类型转换外，程序员可以为用户自定义类型定义新的转换规则。接收单个参数的构造函数就定义了一种从参数类型到其自身类型的转换。如果构造函数是显式的（见 13.4.1 节），只有在程序员显式要求类型转换时这种转换才会进行。否则，转换可以隐式进行。

A.5.3 常量表达式

常量表达式 (constant expression) 是指在编译时即可进行求值的表达式。例如：

```
const int a = 2.7*3;
const int b = a+3;
```

```
constexpr int a = 2.7*3;
constexpr int b = a+3;
```

一个 `const` 可以用包含变量的表达式初始化。而一个 `constexpr` 只能用一个常量表达式初始化。某些程序结构需要使用常量表达式，如数组大小、`case` 语句标号、枚举量初始化代码、`int` 模板参数等等，例如：

```
int var = 7;
switch (x) {
    case 77:      // 正确
    case a+2:     // 正确
    case var:     // 错误 (var 不是一个常量表达式)
        ...
};
```

声明为 `constexpr` 的函数可用于常量表达式。

A.5.4 sizeof

在 `sizeof(x)` 中，`x` 可以是一个类型，也可以是一个表达式。如果 `x` 是一个表达式，

`sizeof(x)` 的值是结果对象的大小。如果 `x` 是一个类型，`sizeof(x)` 的值是一个类型为 `x` 的对象的大小。大小都是以字节来计量的。根据定义，`sizeof(char)==1`。

A.5.5 逻辑表达式

C++ 为整数类型提供了逻辑运算符：

位运算符	
<code>x&y</code>	<code>x</code> 和 <code>y</code> 的位与
<code>x y</code>	<code>x</code> 和 <code>y</code> 的位或
<code>x^y</code>	<code>x</code> 和 <code>y</code> 的位异或

逻辑运算符	
<code>x&&y</code>	<code>x</code> 和 <code>y</code> 的逻辑与，返回 <code>true</code> 或 <code>false</code> ，只有 <code>x</code> 为真时才对 <code>y</code> 求值
<code>x y</code>	<code>x</code> 和 <code>y</code> 的逻辑或，返回 <code>true</code> 或 <code>false</code> ，只有 <code>x</code> 为假时才对 <code>y</code> 求值

位运算符对运算对象逐位进行计算，而逻辑运算符（`&&` 和 `||`）将 0 作为 `false` 处理，将其他值都作为 `true`。位运算法则定义如下：

&	0	1		0	1	^	0	1
0	0	0	0	0	1	0	0	1
1	0	1	1	1	1	1	1	0

A.5.6 new 和 delete

自由空间（动态空间、堆）中的内存是通过 `new` 来分配，通过 `delete`（对单个对象）和 `delete[]`（对数组）来释放的。如果内存已经耗尽，`new` 会抛出一个 `bad_alloc` 异常。如果 `new` 操作成功，它至少为对象分配一个字节的内存空间，并返回其指针。对象的类型是在 `new` 操作之后指定的。例如：

```
int* p1 = new int;           // 分配一个(未初始化的)int
int* p2 = new int[7];         // 分配一个初始化为7的int
int* p3 = new int[100];        // 分配100个(未初始化的)int
...
delete p1;                  // 释放单个对象
delete p2;
delete[] p3;                // 释放数组
```

如果用 `new` 分配的是内置类型对象，除非你明确给出初始化代码，否则不会对对象进行初始化。如果用 `new` 为类对象分配内存，而类具有构造函数，则构造函数将被调用。除非你指明初始化代码，否则将会调用默认构造函数（见 12.4.4 节）。

`delete` 会对其操作对象调用析构函数（如果存在的话）。注意，析构函数可能是虚函数（见附录 A.12.3.1）。

A.5.7 类型转换

C++ 中有四种类型转换运算符：

类型转换运算符

<code>x=dynamic_cast<D*>(p)</code>	尝试将 p 转换为 D* 类型 (可能返回 0)
<code>x=dynamic_cast<D&>(*p)</code>	尝试将 *p 转换为 D& 类型 (可能抛出 <code>bad_cast</code> 异常)
<code>x=static_cast<T>(v)</code>	如果类型为 T 的对象可以转换为 v 的类型，则将 v 转换为 T 类型
<code>x=reinterpret_cast<T>(v)</code>	将 v 转换为位模式一致的类型 T 对象
<code>x=const_cast<T>(v)</code>	将 v 转换为类型 T，添加或去除 <code>const</code> 限定
<code>x=(T)v</code>	C 风格转换：进行任意的老式类型转换
<code>x=T(v)</code>	函数式转换：进行任意的老式类型转换
<code>X=T(v)</code>	从 v 构造一个 T (不进行窄化)

动态转换运算符通常用于在类层次中进行类型转换：p 是指向基类对象的指针，而 D 是派生类。如果 p 不是 D* 类型的话，会返回 0。如果你希望 `dynamic_cast` 在此情况下抛出异常 (`bad_cast`) 而不是返回 0，那么应该使用引用转换代替指针转换。动态类型转换是唯一依赖运行时类型检查的类型转换操作。

静态转换运算符用于“恰当的、行为良好的类型转换”，即 v 原本就是从类型为 T 的对象通过隐式类型转换而得到的。参见 12.8 节。

重解释转换用于将位模式解释为另一种类型。这种类型转换不保证是可移植的，实际上，最好假定每个 `reinterpret_cast` 都是不可移植的。一个典型的重解释转换的例子是 int 到指针的转换，以获取内存地址，参见 12.8 节和 25.4.1 节。

C 风格和函数式转换可以实现 `static_cast` 或 `reinterpret_cast` 与 `const_cast` 组合所能做的任何类型转换。

在程序中最好避免进行类型转换。在大多数情况下，应将其看作糟糕程序设计的标志。当然也有例外情况，12.8 节和 25.4.1 节中对此进行了介绍。C 风格转换和函数式转换具有一些难以理解的特性，你不必彻底弄清涉及这些转换过程的细节内容（见 27.3.4 节）。如果显式类型转换是不可避免的，请使用命名的类型转换。

A.6 语句

下面是 C++ 语句的文法 (*opt* 意为“可选的”):

```

statement:
    declaration
    { statement-listopt }
    try { statement-listopt } handler-list
    expressionopt ;
    selection-statement
    iteration-statement
    labeled-statement
    control-statement

selection-statement:
    if ( condition ) statement
    if ( condition ) statement else statement
    switch ( condition ) statement

iteration-statement:
    while ( condition ) statement
    do statement while ( expression );
    for ( for-init-statement conditionopt; expressionopt ) statement
    for ( declaration : expression ) statement
  
```

labeled-statement:

```
case constant-expression : statement
default : statement
identifier : statement
```

control-statement:

```
break ;
continue ;
return expressionopt ;
goto identifier ;
```

statement-list:

```
statement statement-listopt
```

condition:

```
expression
type-specifier declarator = expression
```

for-init-statement:

```
expressionopt ;
type-specifier declarator = expression ;
```

handler-list:

```
catch ( exception-declaration ) { statement-listopt }
handler-list handler-listopt
```

注意，声明也是语句，而且 C++ 中不存在赋值语句或者过程调用语句——赋值和函数调用都是表达式。更多的内容请参考：

- 迭代语句（`for` 和 `while`）参见 4.4.2 节。
- 选择语句（`if`、`switch`、`case` 和 `break`）参见 4.4.1 节。`break` 语句“跳出”它所在的最内层的 `switch`、`while`、`do` 或者 `for` 语句，即下一条将要执行的语句是跟随在包含 `break` 的结构之后的语句。
- 表达式参见附录 A.5 和 4.3 节。
- 声明参见附录 A.5 和 8.2 节。
- 异常（`try` 和 `catch`）参见 5.6 和 14.4 节。

下面是一个简单的例子，用来说明不同类型的语句（这段程序完成什么功能？）：

```
int* f(int p[], int n)
{
    if (p==0) throw Bad_p(n);
    vector<int> v;
    int x;
    while (cin>>x) {
        if (x==terminator) break; // 退出 while 循环
        v.push_back(x);
    }
    for (int i = 0; i<v.size() && i<n; ++i) {
        if (v[i]==*p)
            return p;
        else
            ++p;
    }
    return 0;
}
```

A.7 声明

声明由三部分组成：

- 要声明的实体的名字；
- 要声明的实体的类型；
- 要声明的实体的初值（大多数情况下是可选的）。

我们可以声明下列实体：

- 内置类型对象和自定义类型对象（见附录 A.8）；
- 用户自定义类型（类和枚举）（见附录 A.10 ~ A.11 及第 9 章）；
- 模板（类模板和函数模板）（见附录 A.13）；
- 别名（参见录 A.16）；
- 名字空间（见附录 A.15 和 8.7 节）；
- 函数（包括成员函数和运算符）（见附录 A.9 和第 8 章）；
- 枚举量（枚举类型的值）（见附录 A.11 和 9.5 节）；
- 宏（见附录 A.17.2 和 27.8 节）。

初始化器可以是一个 {} 包围的表达式列表，包含零个或多个元素（见 3.9.2 节、9.4.2 节和 13.2 节）。例如：

```
vector<int> v {a,b,c,d};
int x {y*z};
```

如果定义中给出的对象类型为 **auto**，则必须进行对象初始化，从而对象类型被确定为初始化器的类型（见 13.3 和 21.2 节）。例如：

```
auto x = 7;           // x 是一个 int
const auto pi = 3.14; // pi 是一个 double
for (const auto& x : v) // x 是指向元素 v 的引用
```

A.7.1 定义

如果一个声明进行初始化、分配内存空间，或者以其他方式提供在程序中使用名字所有的必要信息，则称之为定义（definition）。每个类型、对象和函数都应该在程序中进行一次，且只进行一次定义。例如：

```
double f();           // 一个声明
double f() /* ... */; // (也是) 一个声明
extern const int x;   // 一个声明
int y;                // (也是) 一个定义
int z = 10;            // 一个定义，带显式初始化器
```

常量（**const**）必须进行初始化，因此要求 **const** 的声明中必须带有初始化器，除非显式使用了 **extern**（意味着其带有初始化器的定义必在其他某处）或者其类型具有默认构造函数（见附录 A.12.3）。类的 **const** 成员必须在每个构造函数中都使用成员初始化列表进行初始化（见附录 A.12.3）。

A.8 内置类型

C++ 支持许多基本类型，以及对基本类型应用修饰符构造出的类型：

内置类型

bool x	x 是布尔类型（值为 true 和 false）。
char x	x 是字符类型（通常宽度为 8 位）。
short x	x 是短整型（通常为 16 位）。
int x	x 是默认的整数类型。
float x	x 是浮点数（“短双精度”）。
double x	x 是（“双精度”）浮点数。
void* p	p 是指向裸内存（未知类型的内存）的指针。
T* p	p 是类型 T 的指针。
T *const p	p 是类型 T 的常量（不可变）指针。
T a[n]	a 是 n 个类型为 T 的对象的数组。
T& r	r 是类型 T 的引用。
T f(arguments)	f 是接受 arguments 参数，返回 T 的函数。
const T x	x 是类型 T 的常量（不可变）。
long T x	x 的类型是 long T
unsigned T x	x 的类型是 unsigned T
signed T x	x 的类型是 signed T

这里，T 表示“某种类型”，因此我们可以构造出 long unsigned int、long double、unsigned char 以及 const char *（指向常量字符的指针）等实际类型。然而，这个类型系统并非一个完美的通用系统。例如，不存在 short double（等同于 float），不存在 signed bool（无意义），不存在 short long int（short long 是冗余的），也没有 long long long long int。一个 long long 保证包含至少 64 位。

C++ 支持的浮点类型（floating-point type）包括 float、double 和 long double，它们实际上都是实数的近似。

整数类型（integer type，有时也被称为 integral type）包括 bool、char、short、int、long 和 long long，以及这些类型的无符号的变形。注意，在需要用到整数类型和整型值的地方，通常可以使用枚举类型和枚举值。

我们已经在 3.8 节、12.3.1 节和 25.5.1 节中讨论了内置类型的大小，在第 12 和 13 章中讨论了指针和数组，在 8.5.4 ~ 8.5.6 节中讨论了引用。

A.8.1 指针

指针（pointer）是对象或者函数的地址。指针保存在指针类型的变量中。合法的指针中保存着对象的地址：

```
int x = 7;
int* pi = &x;           // pi 指向 x
int xx = *pi;          // *pi 为 pi 指向的对象的值，在本例中为 7
```

非法指针是指指针中保存的不是任何一个对象的地址：

```
int* pi2;              // 未初始化
*pi2 = 7;              // 未定义行为
pi2 = nullptr;         // 空指针（pi2 仍无效）
*pi2 = 7;              // 未定义行为

pi2 = new int(7);      // now pi2 is valid
int xxx = *pi2;        // fine: xxx becomes 7
```

我们应该尽力使非法指针中保存空指针 (`nullptr`)，这样就可以很容易地检测非法指针：

```
if (p2 == nullptr) { // “若无效”
    // 不要使用 *p2
}
```

更简单的形式：

```
if (p2) { // “若有效”
    // 使用 *p2
}
```

有关指针的更多内容请参考 12.4 节和 13.6.4 节。

下表列出了（非 `void`）对象指针所支持的运算：

指针运算	
<code>*p</code>	解引用 / 间接寻址
<code>p[i]</code>	解引用 / 数组下标
<code>p=q</code>	指针赋值和初始化
<code>p==q</code>	指针相等判定
<code>p!=q</code>	指针不相等判定
<code>p+i</code>	加整数
<code>p-i</code>	减整数
<code>p-q</code>	距离：指针减法
<code>++p</code>	前增 1 (前移)
<code>p++</code>	后增 1 (前移)
<code>--p</code>	前减 1 (后移)
<code>p--</code>	后减 1 (后移)
<code>p+=i</code>	前移 i 个元素
<code>p-=i</code>	后移 i 个元素

注意，只有在指针指向数组内时，才可以进行指针的算术运算（如 `++p` 和 `p+=7`）。对指向数组边界之外的指针进行解引用，其效果是未定义的（编译器或者语言的运行时系统也很可能不会检查这种情况）。我们可对相同类型且指向相同对象或数组的指针进行比较运算 `<`、`<=`、`>` 和 `>=`。

对 `void*` 指针只能进行拷贝（赋值或者初始化）、类型转换运算和比较运算（`==`、`!=`、`<`、`<=`、`>` 和 `>=`）。

函数指针（见 27.2.5 节）只能进行拷贝和调用。例如：

```
using Handle_type = void (*)(int);
void my_handler(int);
Handle_type handle = my_handler;
handle(10); // 等价于 my_handler(10)
```

A.8.2 数组

数组（array）是一个固定长度的、连续的、给定类型对象（元素）的序列：

```
int a[10]; // 10 个 int
```

如果数组是全局的，在定义时，其元素会根据给定类型被初始化为适当的默认值。例如，上面程序中 `a[7]` 的元素会被初始化为 0。如果数组是局部的（在函数中声明），或者是用 `new` 分配的，那么如果元素类型是内置，则不会被初始化，如果元素类型是类，则会通过类

的构造函数进行初始化。

数组名可以隐式转换为指向首元素的指针。例如：

```
int* p = a; // p 指向 a[0]
```

数组或者指向数组元素的指针可以使用下标运算符[]，例如：

```
a[7] = 9;
int xx = p[6];
```

数组元素从 0 开始编号，参见 13.6 节。

数组不进行范围检查，而且由于数组常常以指针方式作为参数传递，即使我们希望自己编写代码进行范围检查，也很难获得可靠的相关信息。如果希望进行类型检查，请使用 `vector`。

数组的大小等于其元素大小之和，例如：

```
int a[max]; // sizeof(a), 即 sizeof(int)*max
```

你可以定义、使用数组的数组（二维数组），数组的数组的数组……（多维数组）。例如：

```
double da[100][200][300]; // 100 个元素, 类型为
                           // 200 个元素, 类型为
                           // 100 个 double
da[7][9][11] = 0;
```

较为复杂的多维数组有很多微妙的细节，也容易出错，参见 24.4 节。如果可以选择的话，最好使用一个 `Matrix` 库（如第 24 章中的 `Matrix`）来实现多维数组。

A.8.3 引用

引用（reference）是对象的别名（可替代的名字）：

```
int a = 7;
int& r = a;
r = 8; // a 变为 8
```

引用常用于函数参数，目的是避免拷贝：

```
void f(const string& s);
...
f("this string could be somewhat costly to copy, so we use a reference");
```

参见 8.5.4 ~ 8.5.6 节。

A.9 函数

函数（function）是命名的代码片段，接受一组参数（可以为空），可以返回一个值，也可以不返回。函数声明由返回类型后接函数名再接参数列表组成：

```
char f(string, int);
```

因此，`f` 是接受一个 `string` 和一个 `int`，返回一个 `char` 的函数。如果仅仅是函数声明，在这些内容之后以一个分号结束。如果是函数定义，在参数列表之后还有函数体：

```
char f(string s, int i) { return s[i]; }
```

函数体必须是一个语句块（见 8.2 节）或者一个 `try` 块（见 5.6.3 节）。

如果声明中指出函数返回一个值，则在函数体中必须用 `return` 语句返回值：

```
char f(string s, int i) { char c = s[i]; } // 错误：未返回值
```

main 函数是一个奇怪的例外（见附录 A.1.2）。除 main() 之外，如果你不希望一个函数返回值，应将其声明为 void 类型，即，“返回类型”为 void：

```
void increment(int& x) { ++x; } // 正确：不需要返回值
```

函数调用通过调用运算符（应用运算符）“()”来实现，应在括号中给出恰当的参数列表：

```
char x1 = f(1,2); // 错误：f() 的第一个参数必须是一个 string
string s = "Battle of Hastings";
char x2 = f(s); // 错误：f() 要求两个参数
char x3 = f(s,2); // 正确
```

更多函数相关的内容请参考第 8 章。

函数定义可以 constexpr 为前缀。此时，当用常量表达参数调用函数时，必须令编译器对函数的求值足够简单。constexpr 函数可用于常量表达式中（见 8.5.9 节）。

A.9.1 重载解析

重载解析（overload resolution）是指根据实参集合选择恰当的函数进行调用的过程。例如：

```
void print(int);
void print(double);
void print(const std::string&);

print(123); // 使用 print(int)
print(1.23); // 使用 print(double)
print("123"); // 使用 print(const string&)
```

根据语言规则选择正确的函数是编译器的任务。不幸的是，为了处理复杂的代码，相关的语言规则相当复杂。本小节给出简化后的规则。

从一组重载函数中选择正确的版本进行调用，是通过寻找实参表达式类型和函数形参类型的最佳匹配实现的。按顺序应用如下解析规则，就可以逐步逼近最合理的结果：

- 精确匹配，即不使用类型转换，或者只需最简单的类型转换（如数组名到指针的转换，函数名到函数指针的转换，T 到 const T 的转换），就能达到匹配。
- 使用提升后匹配，即整型提升（bool 转换为 int，char 转换为 int，short 转换为 int，以及对应的无符号转换，参见附录 A.8）以及 float 转换为 double。
- 使用标准类型转换后达到匹配，例如，int 转换为 double、double 转换为 int、double 转换为 long double、派生类指针转换为基类指针（见 19.3 节）、T* 转换为 void*（见 12.8 节）以及 int 转换为 unsigned int（见 25.5.3 节）。
- 使用用户自定义类型转换后达到匹配（见附录 A.5.2.3）。
- 在函数声明中使用了参数列表省略 “...”（见附录 A.9.3），在此情况下达到匹配。

如果在找到匹配的最高级别上有不只一个匹配，则函数调用因为产生歧义而失败。这套精致的解析规则主要考虑的是内置的数值类型（见附录 A.5.3）。

对于基于多参数的重构解析，我们首先为每个参数寻找最佳匹配。如果一个函数在每个参数的匹配上都至少与其他函数一样好，而且在一个参数的匹配上是最好的，就选择这个函数。如果找不到这样的函数，调用就是有歧义的。例如：

```
void f(int, const string&, double);
void f(int, const char*, int);
```

```
f(1,"hello",1);           // 正确: 调用 f(int, const char*, int)
f(1,string("hello"),1.0); // 正确: 调用 f(int, const string&, double)
f(1, "hello",1.0);       // 错误: 二义性
```

在最后一个调用中，“hello”无须转换就与 `const char*` 匹配，只需一次转换就可与 `const string&` 匹配。另一方面，1.0 无须转换就与 `double` 匹配，只需一次转换就与 `int` 匹配。因此，任何一个 `f()` 都不能得到比其他函数更好的匹配。

如果这些简化的规则与你的编译器所使用的规则不一致，或者与你头脑中合理的规则不一致，请先考虑是不是你的程序过于复杂了。如果是这样的话，请简化你的程序；如果不是这样，请查阅专家级别的参考资料。

A.9.2 默认参数

通用函数通常要使用比一般函数多得多的参数。为了解决这个问题，程序员可以将一些参数设置为默认参数，如果调用者未指定这些参数的值，则使用默认参数值。例如：

```
void f(int, int=0, int=0);
f(1,2,3);
f(1,2);    // 调用 f(1,2,0)
f(1);      // 调用 f(1,0,0)
```

只有末尾的参数才能设置为默认参数，才能在调用时省略。例如：

```
void g(int, int =7, int); // 错误: 只有末尾参数才能设置为默认
f(1,);                  // 错误: 缺失第二个参数
```

函数重载可以作为默认参数的替代方法（反之亦然）。

A.9.3 未指定参数

在函数声明中可以不指明参数的类型，而是使用参数省略符号（`...`），它表示“可能有更多的参数”。最著名的例子是 C 函数 `printf`（见 27.6.1 节和附录 C.11.2），下面是其声明和调用的示意：

```
void printf(const char* format ...); // 接受一个格式字符串和可能更多的参数

int x = 'x';
printf("hello, world!");
printf("print a char '%c\n'",x);      // 将 int 变量 x 作为 char 打印
printf("print a string \"%s\"",x);     // 搬起石头砸自己的脚
```

格式字符串中的“格式限定符”（如 `%c`、`%s`）决定了是否还使用后面的参数，以及如何使用。如上例所示，这可能导致棘手的类型错误。在 C++ 程序中，最好避免使用未指定参数。

A.9.4 连接规范

C++ 代码常常与 C 代码混合使用，即程序的某些部分是用 C++ 编写的（用 C++ 编译器编译），而其他部分是用 C 编写的（用 C 编译器编译）。为了方便这种开发方式，C++ 提供了连接规范（linkage specification）特性，程序员可以用来指明函数遵守 C 的连接约定。连接规范一般置于函数声明的最前面：

```
extern "C" void callable_from_C(int);
```

你也可以用连接规范指定一个块内的所有函数声明均遵守 C 的连接约定：

```
extern "C" {
    void callable_from_C(int);
    int and_this_one_also(double, int*);
    /* ... */
}
```

更多细节参见 27.2.3 节。

C 不支持函数重载，因此，对 C++ 的一组重载函数，你最多只能对其中一个函数使用连接规范。

A.10 用户自定义类型

C++ 中有两种定义新的（用户自定义）类型的方法：类（class、struct 或 union，参见附录 A.12）和枚举（enum，参见附录 A.11）。

A.10.1 运算符重载

对于大多数运算符，程序员都可以重新定义其意义，运算对象可以是一个或多个用户自定义类型的对象。对于内置类型，C++ 不允许改变运算符的标准意义，C++ 也不支持引入新的运算符。用户自定义运算符（“重载运算符”）的名字由运算符加上关键字 operator 前缀组成，例如，定义 + 的函数名为 operator +：

```
Matrix operator+(const Matrix&, const Matrix&);
```

更多的例子，请参考 std::ostream（第 10、11 章）、std::vector（第 12 ~ 14 章，附录 C.4）、std::complex（附录 C.9.3）和 Matrix（第 24 章）。

除了下列运算符之外，其他所有运算符均可重载：

```
? : . .* :: sizeof typeid alignas noexcept
```

定义下列运算符的函数必须是类的成员函数：

```
= [] () ->
```

所有其他可重载的运算符既可以定义为成员函数，也可以定义为独立函数。

注意，每个用户自定义类型都有默认的重载运算符 =（赋值和初始）、&（地址）和 ,（分号）。使用运算符重载功能一定要谨慎且遵循惯例。

A.11 枚举

枚举（enumeration）定义一组命名的值（枚举量，enumerator）：

```
enum Color { green, yellow, red };           // “普通” 枚举
enum class Traffic_light { yellow, red, green }; // 限域枚举
```

enum class 的枚举量位于枚举类型的作用域中，而一个“普通” enum 的枚举量则暴露在 enum 所在的作用域中。例如：

```
Color col = red; // 正确
Traffic_light tl = red; // 错误：不能将整型值（即 Color::red）转换为 Traffic_light 类型
```

默认情况下，第一个枚举量为 0，因此 Color::green==0；随后每个枚举量为前一个枚举量增 1，因此 yellow==1, red==2。你也可以显式定义枚举量的值：

```
enum Day { Monday=1, Tuesday, Wednesday };
```

这段代码中，Monday==1，Tuesday==2，Wednesday==3。

枚举量和“普通”enum的枚举值可以隐式地转换为整数，但整数不能隐式地转换为枚举类型：

```
int x = green;           // 正确：隐式地将 Color 转换为 int
Color c = green;         // 正确
c = 2;                  // 错误：int 不能隐式转换为 Color
c = Color(2);           // 正确：(未检查的) 显式转换
int y = c;               // 正确：Color 到 int 的隐式转换
```

枚举和enumclass的枚举值不可以转换为整数，且整数也不能隐式地转换为枚举类型：

```
int x = Traffic_light::green; // 错误：Traffic_light 不能隐式地转换为 int
Traffic_light c = green;     // 错误：int 不能隐式地转换为 Traffic_light
```

枚举类型使用的内容，请参考9.5节。

A.12 类

类(class)是一种数据类型，用户用来定义对象如何表示、允许哪些操作：

```
class X {
public:
    // 用户接口
private:
    // 实现
};
```

类声明中定义的变量、函数或类型称为类的成员(member)。类的技术细节参见第9章。

A.12.1 成员访问

公有(public)成员可被类的使用者访问，私有(private)成员只能被类的成员访问：

```
class Date {
public:
    // ...
    int next_day();
private:
    int y, m, d;
};

void Date::next_day() { return d+1; }      // 正确

void f(Date d)
{
    int nd = d.d+1;           // 错误：Date::d 是私有的
    // ...
}
```

struct是特殊的类，默认情况下，其成员都是公有的：

```
struct S {
    // 成员（除非显式说明是私有的，否则是默认公有的）
};
```

成员访问的更多细节，包括protected的有关内容，参见19.3.4节。

对象的成员通过对变量或其引用使用.(点)运算符，或者对其指针使用->(箭头)运算符来访问：

```

struct Date {
    int d, m, y;
    int day() const { return d; }      // 定义在类内
    int month() const;                // 只是声明；定义在别处
    int year() const;                 // 只是声明；定义在别处
};

Date x;
x.d = 15;                           // 通过变量访问
int y = x.day();                    // 通过变量调用
Date* p = &x;
p->m = 7;                          // 通过指针访问
int z = p->month();                // 通过指针调用

```

类的成员可以通过 :: (作用域解析) 运算符来引用：

```
int Date::year() const { return y; } // 类外定义
```

在成员函数内访问其他成员，可以直接使用未限定的名字：

```

struct Date {
    int d, m, y;
    int day() const { return d; }
    ...
};

```

这些未限定的名字引用的是调用此成员函数所用的对象的成员：

```

void f(Date d1, Date d2)
{
    d1.day();    // 将访问 d1.d
    d2.day();    // 将访问 d2.d
    ...
}

```

A.12.1.1 this 指针

如果希望在成员函数中显式引用所用对象，可以使用预定义的指针 this：

```

struct Date {
    int d, m, y;
    int month() const { return this->m; }
    ...
};

```

声明为 const 的成员函数 (const 成员函数) 不能修改调用对象的成员的值：

```

struct Date {
    int d, m, y;
    int month() const { ++m; } // 错误：month() 是 const
    ...
};

```

有关 const 成员函数的更多内容，参见 9.7.4 节。

A.12.1.2 友元函数

我们可以通过 friend 声明来授予独立函数访问所有类成员的权限，例如：

```

// 需要访问 Matrix 和 Vector 成员
Vector operator*(const Matrix&, const Vector&);

class Vector {
    friend

```

```

Vector operator*(const Matrix&, const Vector&); // 授权访问
// ...
};

class Matrix {
    friend
        Vector operator*(const Matrix&, const Vector&); // 授权访问
    // ...
};

}

```

如这段代码所示，**friend** 通常用于那些需要访问两个类的函数，另一个用途是那些无法使用成员函数调用语法的函数。例如：

```

class Iter {
    public:
        int distance_to(const iter& a) const;
        friend int difference(const Iter& a, const Iter& b);
        // ...
};

void f(Iter& p, Iter& q)
{
    int x = p.distance_to(q); // 使用成员语法调用
    int y = difference(p,q); // 使用“数学语法”调用
    // ...
}

```

注意，友元函数不能是虚函数。

A.12.2 类成员定义

如果类成员是整型常量、函数或者类型，则既可在类内（in-class）定义和初始化（见 9.7.3 节），也可在类外（out-of-class）定义和初始化（见 9.4.4 节）：

```

struct S {
    int c = 1;
    int c2;

    void f() {}
    void f2();

    struct SS { int a; };
    struct SS2;
};

```

未在类内定义的成员必须在“其他某处”定义：

```

int S::c2 = 7;

void S::f2() {}

struct S::SS2 { int m; };

```

如果你希望用对象创建者指定的值来初始化一个数据成员，则应在其构造函数内进行。函数成员不占用对象空间：

```

struct S {
    int m;
    void f();
};

}

```

在这段代码中，`sizeof(S)==sizeof(int)`。C++ 标准并不保证这一特性，但实际上我们所知道的所有 C++ 编译器都满足这一特性。但请注意，如果类中定义了虚函数，则有一个“隐藏的”成员用来解析虚拟函数调用（见 19.3.1 节）。

A.12.3 构造、析构和拷贝

你可以通过定义一个或多个构造函数（constructor）来定义对象初始化操作。构造函数也是一种成员函数，只是其名字与类名相同，且无返回类型：

```
class Date {
public:
    Date(int yy, int mm, int dd) :y{yy}, m{mm}, d{dd} {}
    // ...
private:
    int y, m, d;
};

Date d1 {2006, 11, 15}; // 正确：构造函数完成初始化
Date d2; // 错误：无初始化器
Date d3 {11, 15}; // 错误：错误的初始化器（需要三个初始化器）
```

注意，数据成员可以通过构造函数中的初始化器列表（包括基类和成员的初始化器列表）来进行初始化。成员将会按照它们在类声明中的顺序进行初始化。

构造函数通常用来建立类的不变式及分配所需资源（见 9.4.2 和 9.4.3 节）。

类对象是按照一种“自底向上”的方式构造的：首先按声明顺序构造基类对象（见 19.3.1 节），然后按声明顺序构造成员，最后执行自身构造函数。除非程序员做了什么非常奇怪的事情，否则这样的构造过程会保证所有对象都是先构造后使用。

除非显式声明，否则单参数的构造函数定义了此参数到类之间的类型转换。

```
class Date {
public:
    Date(const char*); // 使用一个编码了 Date 的整数
    explicit Date(long); // 使用一个编码了 Date 的整数
    // ...
};

void f(Date);

Date d1 = "June 5, 1848"; // 正确
f("June 5, 1848"); // 正确

Date d2 = 2007*12*31+6*31+5; // 错误：Date(long) 是显式的
f(2007*12*31+6*31+5); // 错误：Date(long) 是显式的

Date d3(2007*12*31+6*31+5); // 正确
Date d4 = Date{2007*12*31+6*31+5}; // 正确
f(Date{2007*12*31+6*31+5}); // 正确
```

除非类包含需要显式参数的基类或成员，或者已经定义了其他构造函数，否则编译器会自动为其生成一个默认构造函数。这个默认构造函数会初始化所有具有默认构造函数的基类和成员（忽略没有默认构造函数的成员）。例如：

```
struct S {
    string name, address;
```

```

    int x;
};

}

```

S 具有一个隐式构造函数 S(), 它初始化 name 和 address，但不会初始化 x。此外，没有构造函数的类可以用初始化器列表进行初始化：

```

S s1 {"Hello!"};           // s1 变为 {"Hello!", 0}
S s2 {"Howdy!", 3};
S* p = new S("G'day!");   // *p 变为 {"G'day", 0}

```

如上面代码所示，未指定的末尾参数变为默认值（本例中是将 0 赋予 int）。

A.12.3.1 析构函数

你可以通过定义析构函数（destructor）来定义对象销毁操作（如对象超出作用域时）。析构函数的名字由一个 ~（补运算符）后接类名构成：

```

class Vector { // double 的 vector
public:
    explicit Vector(int s) : sz(s), p{new double[s]} {} // 构造函数
    ~Vector() { delete[] p; }                            // 析构函数
    ...
private:
    int sz;
    double* p;
};

void f(int ss)
{
    Vector v(ss);
    ...
} // 当从 f() 退出时，v 会被销毁；会对 v 调用 Vector 的析构函数

```

编译器可以自动生成析构函数，这个析构函数调用每个类成员的析构函数。如果类为基类，则通常需要一个虚析构函数，参见 12.5.2 节。

析构函数通常用来做“清理”工作及释放资源。

类对象是按一种“自顶向下”的方式析构的：首先执行析构函数，然后按声明顺序析构类成员，最后按声明顺序析构基类对象，即与构造过程完全相反的顺序。

A.12.3.2 拷贝

你可以为类对象定义拷贝操作（copying）：

```

class Vector { // double 的 vector
public:
    explicit Vector(int s) : sz(s), p{new double[s]} {} // 构造函数
    ~Vector() { delete[] p; }                            // 析构函数
    Vector(const Vector&);                            // 拷贝构造函数
    Vector& operator=(const Vector&);                // 拷贝赋值操作
    ...
private:
    int sz;
    double* p;
};

void f(int ss)
{
    Vector v(ss);
    Vector v2 = v;          // 使用拷贝构造函数
    ...
}

```

```

v = v2;           // 使用拷贝赋值操作
// ...
}

```

默认情况下（即你没有定义拷贝构造函数以及拷贝赋值操作），编译器会为你生成拷贝操作。默认拷贝语义是逐成员拷贝，参见 19.2.4 节和 13.3 节。

A.12.3.3 移动

你可以定义类对象的移动 (moving) 操作的含义：

```

class Vector { // double 的 vector
public:
    explicit Vector(int s) : sz{s}, p{new double[s]} {} // 构造函数
    ~Vector() { delete[] p; } // 析构函数
    Vector(Vector&&); // 移动构造函数
    Vector& operator=(Vector&&); // 移动赋值操作
    // ...
private:
    int sz;
    double* p;
};

Vector f(int ss)
{
    Vector v(ss);
    // ...
    return v; // 使用移动构造函数
}

```

默认情况下（即你没有定义移动构造函数以及移动赋值操作），编译器会为你生成移动操作。默认移动语义是逐成员移动，参见 13.3.4 节。

A.12.4 派生类

一个类可以定义为其他类的派生类，它会继承派生出它的类（基类）的成员：

```

struct B {
    int mb;
    void fb();
};

class D : B {
    int md;
    void fd();
};

```

这段代码中，B 有两个成员 mb 和 fb()，D 有四个成员 mb、fb()、md 和 fd()。

与成员类似，基也可以是 public 或 private：

```

Class DD : public B1, private B2 {
    // ...
};

```

这样，B1 的公有成员都成为 DD 的公有成员，B2 的公有成员都成为 DD 的私有成员。派生类对基类的成员没有特殊的访问权限，因此 DD 不能访问 B1 或 B2 的私有成员。

如果一个类有多个直接基类（如 DD），则称这种继承方式为多重继承 (multiple inheritance)。只要在派生类 D 中，基类 B 是可访问的、无歧义的，那么指向 D 的指针就可以隐式转

换为指向 B 的指针。例如：

```
struct B {};
struct B1: B {};// B 是 B1 的一个公有基类
struct B2: B {};// B 是 B2 的一个公有基类
struct C {};
struct DD : B1, B2, private C {};

DD* p = new DD;
B1* pb1 = p; // 正确
B* pb = p; // 错误：二义性：B1::B 还是 B2::B ?
C* pc = p; // 错误：DD::C 是私有的
```

类似地，派生类的引用也可以隐式转换为无歧义的、可访问的基类的引用。

派生类相关的更多内容请参考 19.3 节。`protected` 相关的更多内容请查阅专家级的教材或参考资料。

A.12.4.1 虚函数

虚函数（virtual function）是一种成员函数，它为派生类中具有相同名字、接受相同参数的函数定义了一个一致的调用接口。当调用一个虚函数时，实际被调用的是对应派生类中的函数。在派生类中定义与基类中虚函数名字和参数相同的函数，被称为虚函数覆盖（override）。

```
class Shape {
public:
    virtual void draw(); // virtual 意味着“可以覆盖”
    virtual ~Shape() {} // 虚析构函数
    ...
};

class Circle : public Shape {
public:
    void draw(); // 覆盖 Shape::draw
    ~Circle(); // 覆盖 Shape::~Shape()
    ...
};
```

大致来说，基类中（本例中是 `Shape`）的虚函数为派生类（本例中是 `Circle`）定义了一个调用接口：

```
void f(Shape& s)
{
    ...
    s.draw();
}

void g()
{
    Circle c{Point{0,0}, 4};
    f(c); // 将会调用 Circle 的 draw
}
```

注意，`f()` 并不知道 `Circle` 类，它只知道 `Shape`。定义了虚函数的类都有一个额外的指针，通过这个指针可以找到覆盖函数，参见 19.3 节。

注意，定义了虚函数的类通常需要定义虚析构函数（如 `Shape`），参见 12.5.2 节。

我们可以用 `override` 后缀表明覆盖基类虚函数的愿望。例如：

```
class Square : public Shape {
public:
    void draw() override; // 覆盖 Shape::draw
    ~Circle() override; // 覆盖 Shape::~Shape()
    void silly() override; // 错误: Shape 没有虚函数 Shape::silly()
    ...
};
```

A.12.4.2 抽象类

抽象类 (abstract class) 是只能用作基类的类, C++ 不允许创建抽象类的对象:

```
Shape s; // 错误: Shape 是抽象类
```

```
class Circle : public Shape {
public:
    void draw(); // 覆盖 Shape::draw
    ...
};
```

```
Circle c{p,20}; // 正确: Circle 不是抽象类
```

使一个类成为抽象类的最常用的方法是定义至少一个纯虚函数。纯虚函数 (pure virtual function) 是必须在派生类中被覆盖的虚函数:

```
class Shape {
public:
    virtual void draw() = 0; // =0 表示“纯”
    ...
};
```

参见 19.3.5 节。

另外一种很少使用但同样有效的定义抽象类的方法是将类的所有构造函数都声明为保护的 (见 19.2.1 节)。

A.12.4.3 默认操作

当你定义一个类时, 编译器可能为类对象定义几个默认操作:

- 默认构造函数;
- 拷贝操作 (拷贝赋值和拷贝初始化);
- 移动操作 (移动赋值和移动初始化);
- 析构函数。

这些 (默认) 操作都会递归地对类的每个基类和成员进行操作。构造是“自底向上”进行的, 即先构造基类, 然后构造成员。析构是“自顶向下”进行的, 即先析构成员, 然后析构基类。成员和基类是按它们在声明中出现的顺序构造的, 按相反的顺序销毁。采用这种方式, 构造函数和析构函数是否正确工作都依赖于定义良好的基类和成员对象。例如:

```
struct D : B1, B2 {
    M1 m1;
    M2 m2;
};
```

假定已经定义了 B1、B2、M1 和 M2, 我们可以编写下面代码:

```
D f()
{
    D d; // 默认初始化
    D d2 = d; // 拷贝初始化
```

```

d = D{};      // 默认初始化，然后拷贝赋值
return d;     // d 移出 f()
} // d 和 d2 被销毁

```

例如，`d` 的默认初始化操作会依次调用四个默认构造函数：`B1::B1()`、`B2::B2()`、`M1::M1()` 和 `M2::M2()`。如果其中一个构造函数不存在，或者无法调用，那么 `d` 的构造操作就会失败。`return` 时依次调用四个移动构造函数：`B1::B()`、`B2::B2()`、`M1::M1(1)` 和 `M2::M2()`。如果其中一个不存在或无法调用，那么 `return` 失败。`d` 的析构操作依次调用四个析构函数：`M2::~M2()`、`M1::~M1()`、`B2::~B2()` 和 `B1::~B1()`。如果其中一个析构函数不存在或者无法调用，`d` 的析构操作就会失败。这些构造函数和析构函数既可以是用户定义的，也可以是编译器自动生成的。

如果程序员已经为类定义了构造函数，则编译器不再自动生成隐式默认构造函数。

A.12.5 位域

位域 (bitfield) 是这样一种机制，它将很多较小的值打包存储在一个字中，或者用来匹配外部位布局格式 (如设备寄存器)。例如：

```

struct PPN { // R6000 物理页编号 (Physical Page Number, PPN)
    unsigned int PFN : 22;      // 页帧编号 (Page Frame Number, PFN)
    int : 3;                  // 未使用
    unsigned int CCA : 3;       // 协同缓存算法 (Cache Coherency Algorithm, CCA)
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};

```

将这些位域打包，由左至右存入一个字中，就得到如下所示的位布局 (见 25.5.5 节)：

位置 :	31:	8:	5:	2: 1: 0:
PPN:	22	3	3	1 1 1
名字 :	PFN	未使用	CCA	脏 全局 有效

位域无须命名，但如果不能命名，就无法访问。

奇怪的是，将很多较小的值打包存入一个字中并不一定能节省内存空间。实际上，与用一个 `char` 或 `int` 描述一个二进制位相比，使用位域方式常常会浪费内存空间。原因在于，从一个字中提取一个二进制位，以及修改字中的一个二进制位而不影响其他位，需要花费好几个机器指令 (而这些指令也是存储在内存中的)。不要试图通过使用位域来节省内存空间，除非你需要大量数据对象，每个对象都包含很小的数据域。

A.12.6 联合

联合 (union) 是一种特殊的类，其所有成员都占用相同的内存地址空间。联合在任何时候都只能保存一个成员，读取的成员必须是最后写入的那个成员。例如：

```

union U {
    int x;
    double d;
}

```

```

U a;
a.x = 7;
int x1 = a.x; // 正确
a.d = 7.7;
int x2 = a.x; // 糟糕

```

编译器并不检查读写的一致性，你必须小心编写程序。

A.13 模板

模板 (template) 是一个类或者一个函数，以一组类型或 / 和整数为参数：

```

template<typename T>
class vector {
public:
    ...
    int size() const;
private:
    int sz;
    T* p;
};

template<class T>
int vector<T>::size() const
{
    return sz;
}

```

在模板参数列表中，class 表示类型，也可以用 typename 表示类型。模板类的成员函数隐含地被定义为模板函数，其模板参数与模板类一致。

整数模板参数必须是常量表达式：

```

template<typename T, int sz>
class Fixed_array {
public:
    T a[sz];
    ...
    int size() const { return sz; };
};

Fixed_array<char,256> x1; // 正确
int var = 226;
Fixed_array<char,var> x2; // 错误：非 const 模板实参

```

A.13.1 模板参数

当使用模板类时，需指定模板参数：

```

vector<int> v1;           // 正确
vector v2;                // 错误：模板实参缺失
vector<int,2> v3;         // 错误：模板实参过多
vector<2> v4;             // 错误：要求模板实参为类型

```

模板函数的模板参数则通常通过函数的实参推断出来：

```

template<class T>
T find(vector<T>& v, int i)
{
    return v[i];
}

```

```

vector<int> v1;
vector<double> v2;
...
int x1 = find(v1,2);      // find() 的 T 为 int
int x2 = find(v2,2);      // find() 的 T 为 double

```

模板函数的模板参数无法通过函数的实参推断出来的情况是完全可能发生的。在这种情况下，我们必须显式指定缺失的模板参数（就像模板类那样）。例如：

```

template<class T, class U> T* make(const U& u) { return new T{u}; }
int* pi = make<int>(2);
Node* pn = make<Node>(make_pair("hello",17));

```

如果 Node 对象可以通过一个 pair<const char *, int>（见附录 C.6.3）进行初始化，这段代码就可以正常工作。当显式指定模板参数时，只允许省略末尾的模板参数（由函数实参推断出来）。

A.13.2 模板实例化

指明了一组特定模板参数的模板版本称为特例（specialization），由模板和一组参数生成特例的过程称为模板实例化（template instantiation）。通常是由编译器从一个模板和一组模板参数生成一个特例，但程序员也可以定义特殊的特例，这通常是因为一般模板不适合一组特殊的模板参数，例如：

```

template<class T> struct Compare {           // 通用比较
    bool operator()(const T& a, const T& b) const
    {
        return a<b;
    }
};

template<> struct Compare<const char*> {    // 比较 C 风格字符串
    bool operator()(const char* a, const char* b) const
    {
        return strcmp(a,b)==0;
    }
};

Compare<int> c2;                           // 通用比较
Compare<const char*> c;                   // C 风格字符串比较
bool b1 = c2(1,2);                         // 使用通用比较
bool b2 = c("asd","dfg");                  // 使用 C 风格字符串比较

```

对于函数，通过重载可以达到大致等价的效果：

```

template<class T> bool compare(const T& a, const T& b)
{
    return a<b;
}

bool compare (const char* a, const char* b) // 比较 C 风格字符串
{
    return strcmp(a,b)==0;
}

bool b3 = compare(2,3);                    // 使用通用比较
bool b4 = compare("asd","dfg");           // 使用 C 风格字符串比较

```

模板的分离编译（即，头文件中只有声明，而唯一的定义置于 .cpp 文件中）不具备可移植性，因此，如果模板需要在多个 .cpp 文件中使用的话，应将其完整定义置于头文件中。

A.13.3 模板成员类型

模板的成员可以是类型，也可以不是类型（如数据成员和成员函数）。这意味着，一般很难分辨一个成员名是类型还是非类型。出于语言技术上的原因，编译器必须要知道这方面的信息，因此有时我们必须将相关信息告知编译器。为此，我们需要使用关键字 `typename`。例如：

```
template<class T> struct Vec {
    typedef T value_type; // 一个成员类型
    static int count; // 一个数据成员
    ...
};

template<class T> void my_fct(Vec<T>& v)
{
    int x = Vec<T>::count; // 默认情况下，编译器假定成员名引用的是非类型成员
    v.count = 7; // 引用非类型成员的更简单的方法
    typename Vec<T>::value_type xx = x; // 这里需要 typename
    ...
}
```

模板的更多内容参见第 14 章。

A.14 异常

异常用来（通过 `throw` 语句）告知调用者，发生了一个局部无法处理的错误。例如，`Vector` 抛出 `Bad_size` 异常：

```
struct Bad_size {
    int sz;
    Bad_size(int s) : ss{s} {}
};

class Vector {
    Vector(int s) { if (s<0 || maxsize<s) throw Bad_size{s}; }
    ...
};
```

通常，我们为每个错误定义专门的异常类型。调用者可以捕获异常：

```
void f(int x)
{
    try {
        Vector v(x); // 可能抛出异常
        ...
    }
    catch (Bad_size bs) {
        cerr << "Vector with bad size (" << bs.sz << ")\n";
        ...
    }
}
```

子句“捕获所有”可以用来捕获所有异常：

```
try {
    //...
} catch (...) { // 捕获所有异常
    //...
}
```

通常，使用 RAII (Resource Acquisition Is Initialization，资源获取即初始化) 技术比使用大量显式的 **try** 和 **catch** 更好，参见 14.5 节。

没有参数的 **throw** (即 **throw;**) 表示重新抛出刚捕获的异常。例如：

```
try {
    //...
} catch (Some_exception& e) {
    // 进行局部清理
    throw; // 让我的调用者进行剩余清理工作
}
```

你可以将自定义类型作为异常使用。标准库中也定义了一些异常类型，参见附录 C.2.1。不要将内置类型用作异常（某些人可能已经这样做了，这会导致你的异常类型被这些不合法的异常弄乱）。

当一个异常被抛出后，C++ 的运行时支持系统“在调用栈中向上”搜索与抛出对象类型匹配的 **catch** 子句，即查找抛出异常的函数中的 **try** 语句，接着在抛出异常的函数的调用者中查找，然后在更上层的调用者中查找，依此类推，直至找到匹配的 **catch** 子句。如果找不到匹配的 **catch** 子句，程序就会终止。在搜索过程中遇到的每个函数中，以及每个作用域中，都会调用析构函数进行清理工作，这个过程被称为堆栈解退 (stack unwinding)。

一旦对象的构造函数执行完毕，我们就认为对象已经构造出来了，而在堆栈解退过程中，或者退出对象作用域时，对象被销毁。这意味着，作用域中还没有完全构造完成的对象（某些成员或基类已经构造完成，而另外一些没有构造完成）、数值和变量能够被正确处理。子对象当然仅当构造完毕后才能销毁。

不要在析构函数中抛出异常，这条原则意味着析构函数不能失败。例如：

```
X::~X() { if (in_a_real_mess() throw Mess{}; } // 永远不要这样做！
```

提出这样一条严厉原则的主要原因是，如果析构函数在堆栈解退过程中抛出一个异常（而它自身又没有捕获这个异常的话），我们不知道应该处理哪个异常。我们应该尽力避免析构函数因为抛出异常而退出，因为在这种情况下可能发生的地方，目前还不知道有什么系统的方法保证写出正确的代码。特别是，如果发生了这种情况，标准库中的特性都无法保证正常工作。

A.15 名字空间

名字空间 (namespace) 将相关的声明组织在一起，用来避免名字冲突：

```
int a;

namespace Foo {
    int a;
    void f(int i)
    {
        a+= i; // 这是 Foo 的 a (Foo::a)
    }
}
```

```

void f(int);

int main()
{
    a = 7;           // 这是全局 a (::a)
    f(2);            // 这是全局 f (::f)
    Foo::f(3);       // 这是 Foo 的 f
    ::f(4);          // 这是全局 f (::f)
}

```

我们可以显式地用名字空间名来限定名字（如 `Foo::f(3)`），或者用 `::` 来限定名字（如 `::f(2)`），后者表示全局作用域。

我们可以使用一条单一的名字空间指令，使名字空间中（在下例中是标准库名字空间，`std`）的所有名字都可被访问：

```
using namespace std;
```

使用 `using` 指令一定要小心，虽然获得了名字使用上的便利性，但可能会导致潜在的名字冲突。特别地，不要在头文件中使用 `using` 指令。我们可以使用名字空间声明，使名字空间中的某个名字可访问：

```

using Foo::g;
g(2);           // 这是 Foo 的 g (Foo::g)

```

有关名字空间的更多内容参见 8.7 节。

A.16 别名

我们可以为名字定义别名（alias），即我们可以定义一个符号名字，它与所引用的内容（大多数情况下）具有完全相同的函数：

```

using Pint = int*;           // Pint 表示 int 指针

namespace Long_library_name { /* ... */}
namespace Lib = Long_library_name; // Lib 表示 Long_library_name

int x = 7;
int& r = x;                 // r 表示 x

```

引用（见 8.5.5 节和附录 A.8.3）是一种运行时机制，用来引用对象。`using`（见 20.5 节）和 `namespace` 别名是编译时机制，引用的是名字。特别是，`using` 并不引入一个新类型，而只是为已有类型定义一个新名字。例如：

```

using Pchar = char*;           // Pchar 是 char* 的一个别名
Pchar p = "ldefix";           // 正确：p 是一个 char*
char* q = p;                 // 正确：p 和 q 都是 char*
int x = strlen(p);             // 正确：p 是一个 char*

```

较旧的代码使用关键字 `typedef`（见 27.3.1 节）而不是（C++）`using` 语法来定义类型别名。例如：

```
typedef char* Pchar;           // Pchar 是 char* 的一个别名
```

A.17 预处理指令

每个 C++ 实现都包含预处理器（preprocessor）。理论上，预处理器在编译器之前运行，

恰当地将我们编写的源码转换为编译器所需要的形式。在实际中，预处理过程通常集成在编译器中，而且除非它引起了错误，否则对我们来说没什么意义。以 # 开头的代码行都是预处理指令。

A.17.1 #include

我们已经大量使用预处理器来包含头文件了。例如：

```
#include "file.h"
```

这条指令告诉预处理器，在源码中指令出现的这个位置包含 file.h 的内容。对于标准头文件，我们使用 <…> 而不是 "...", 例如：

```
#include<vector>
```

这是包含标准头文件的建议语法。

A.17.2 #define

预处理器实现了一种字符处理机制，这种机制被称为宏替换（macro substitution）。例如，我们可以为字符串定义名字：

```
#define FOO bar
```

现在，凡是出现 FOO 的地方，都会被替换为 bar：

```
int FOO = 7;
int FOOL = 9;
```

经过预处理器处理，编译器看到的将是：

```
int bar = 7;
int FOOL = 9;
```

注意，预处理器对 C++ 名字有足够的了解，因此不会将 FOOL 中的 FOO 替换掉。

你还可以定义带参数的宏：

```
#define MAX(x,y) (((x)>(y))?(x):(y))
```

我们可以这样使用这个宏：

```
int xx = MAX(FOO+1,7);
int yy = MAX(++xx,9);
```

这段代码会被扩展为：

```
int xx = (((bar+1)>( 7))?(bar+1):( 7));
int yy = (((++xx)>( 9))?(++xx):( 9));
```

注意括号的必要性，否则 FOO+1 就不会得到正确的结果了。同时请注意，xx “悄悄地” 被做了两次增 1 运算。宏非常普及了，主要原因是，对 C 程序员来说，没有什么有效的替代方法。常用的头文件中定义了数以千计的宏，使用这些宏时一定要小心！

如果你必须要使用宏，被广泛接受的命名习惯是全部使用大写字母。其他普通名字则不应使用全部大写字母，这样就可以避免冲突。当然，不要指望其他人也能遵循这条合理的建议。例如，我们曾经发现在一个颇有影响的头文件中定义了一个名为 max 的宏。

有关宏的更多内容参见 27.8 节。

Visual Studio 简要入门教程

宇宙之神奇不仅超出我们的想象，而且超出我们所能想象。

——J. B. S. Haldane

本附录说明了如何用微软 Visual Studio 来输入一个程序、编译它并使之运行。

B.1 让程序运行起来

为了让程序运行起来，你需要将涉及的文件一起放置在某处（这样当一个文件引用其他文件时——如源程序文件引用头文件——就很容易找到）。然后你需要调用编译器和连接器（如果没有其他程序或库，将程序与 C++ 标准库连接起来），最后你需要运行（执行）程序。完成这样一个过程有很多方法，不同系统（如 Windows 和 Linux）有不同的习惯和工具集。不过，本书中的所有例子，在所有主流系统上，用任何一种主流工具集均可正确编译运行。本附录介绍如何用一个流行的开发系统——微软的 Visual Studio，来完成这些工作。

就个人而言，我发现对少数练习，让它们运行起来和在一个奇怪的新系统上编译运行第一个程序一样困难。这时，最好能寻求帮助。不过，如果你确实寻求到了帮助，不要仅仅让对方帮你完成工作，而应该向其学习如何来做。

B.2 安装 Visual Studio

Visual Studio 是一个 Windows 平台上的交互式开发环境（IDE）。如果你的计算机上还未安装 Visual Studio，你可以购买一份拷贝，按说明进行安装，也可以从 www.microsoft.com/express/download 下载免费的 Visual C++ Express 版。本附录中的描述都是基于 Visual Studio 2005 版的，其他版本有细微差别。

B.3 创建、运行一个程序

步骤如下：

1. 创建一个新项目。
2. 向工程添加一个 C++ 源文件。
3. 输入你的源代码。
4. 生成一个可执行文件。
5. 执行程序。
6. 保存程序。

B.3.1 创建一个新项目

在 Visual Studio 中，“项目”是一个文件集合，这些文件组合在一起，可以在 Windows 平台上创建、运行一个程序（也被称为应用程序（application））。

1. 打开 Visual C++ IDE：点击 Microsoft Visual Studio 2005 图标；或选择“开始”>“程序”>“Microsoft Visual Studio 2005”>“Microsoft Visual Studio 2005”。
2. 打开“文件”菜单，指向“新建”菜单项，然后点击“项目”。
3. 在“项目类型”中选择“Visual C++”。
4. 在“模板”窗口中选择“Win32 控制台应用程序”。
5. 在“名称”文本框输入项目的名字，例如“Hello, World!”。
6. 为项目选择一个目录。缺省目录是“C:\Documents and Settings\Your Name\My Documents\Visual Studio 2005 Projects”，通常使用这个目录就可以了。
7. 点击“确定”。
8. 现在 Win32 应用程序向导对话框应该出现了。
9. 在对话框左侧选择“应用程序设置”。
10. 在“附加选项”栏中选择“空项目”。
11. 点击“完成”。现在你的控制台项目的所有编译器选项应该都已设定好了。

B.3.2 使用 std_lib_facilities.h 头文件

对你的第一个程序，我们强烈建议你使用定制的头文件 std_lib_facilities.h，你可以从这里下载：www.stroustrup.com/Programming/std_lib_facilities.h。将其副本放置于你在 B.3.1 第 6 步中所设定的目录中。（注意：将这个文件保存为文本格式，而不是 HTML 格式。）你需要在源程序中加入下面这行代码来使用这个头文件：

```
#include "../std_lib_facilities.h"
```

“..../”告诉编译器你将头文件放在了目录 C:\Documents and Settings\Your Name\My Documents\Visual Studio 2005 Projects 中，这样，头文件就可以被所有项目使用，而不用为每个项目拷贝一个副本。

B.3.3 向项目中添加 C++ 源文件

在程序中至少需要有一个源文件（通常有很多）：

1. 点击工具条中的“添加新项”图标（通常是左起第二个图标）。这会打开“添加新项”对话框。在“Visual C++”类别下选择“代码”。
2. 在模板窗口中选择“C++ 文件 (.cpp)”图标。在“名称”文本框中输入程序文件的名字（Hello, World!）并点击“添加”。

这样就创建了一个空的源代码文件，现在就可以输入你的源程序了。

B.3.4 输入源码

现在你可以直接在 IDE 中输入源码，也可以从其他源文件中粘贴过来。

B.3.5 生成可执行程序

当你确信已经正确输入源码后，进入“生成”菜单，选择“生成解决方案”菜单项，或者在接近 IDE 窗口顶端的图标列表中点击向右的三角图标。IDE 就会尝试编译并连接你的程序。如果成功，会在“输出”窗口中显示：

Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped

否则就会输出一些错误信息。修改你的程序，去掉这些错误，再尝试“生成解决方案”。

如果你点击了三角图标，在编译成功后，IDE 会自动运行程序。如果使用的是“生成解决方案”菜单项，你必须手工启动程序，如 B.3.6 所述。

B.3.6 执行程序

一旦消除了所有错误，你可以进入“调试”菜单，选择“开始执行（不调试）”菜单项来执行程序。

B.3.7 保存程序

在“文件”菜单下，点击“全部保存”。如果你忘记保存程序就试图关闭 IDE，IDE 会提醒你。

B.4 后续学习

IDE 具有无数特性和选项，不要过早尝试，否则你会彻底迷失。如果你将一个项目搞得“行为怪异”，请向有经验的朋友求助，或者从头开始重建一个新项目。在学习的过程中，你应该慢慢地尝试新的特性和选项。

术语表

通常，精心挑选的几个词就胜过几千幅图。

——匿名

术语表（glossary）是正文中词汇的简单解释。下面是一个非常简短的术语表，列出了我们认为最重要的，特别是在学习编程初期尤为重要的术语。每章的“术语”一节也能帮助你查找术语。更完整的 C++ 相关术语表可在 www.stroustrup.com/glossary.html 找到，在互联网上还能找到非常多的专门的术语表（质量也参差不齐）。请注意，一个术语可能有多个相关的含义（因此我们偶尔可能会列出一些其他含义），而我们列出的大多数术语都在其他场景下有相关的含义（通常是弱相关的）；例如，我们定义抽象（abstract）一词时不会考虑它在现代绘画、法律事务或是哲学中的含义。

abstract class (抽象类) 不能直接用来创建对象的类；通常用来定义派生类的接口。如果一个类具有纯虚函数或保护的构造函数，则它就成为抽象类。

abstraction (抽象) 描述某实体选择性地、故意地忽略（隐藏）细节（如实现细节）；选择性忽略。

address (地址) 一个值，用来在计算机内存中查找一个对象。

algorithm (算法) 求解问题的一个过程或一个公式；一个计算步骤的有限序列，生成一个结果。

alias (别名) 引用一个对象的一种替代方法；通常是一个名字、一个指针或一个引用。

application (应用) 一个程序或一组程序，被其用户看作单一实体。

approximation (近似) 接近最优或理想（值或设计）的东西（如一个值或一个设计）。通常一个近似是理想结果的一个折中。

argument (实参) 传递给函数或模板的值，函数或模板通过参数访问它。

array (数组) 元素的同构序列，元素通常是编号的，如 [0:max]。

assertion (断言) 插入程序中的陈述，声明（断言）在此程序点上某事必须始终为真。

base class (基类) 作为类层次根基的类。通常

一个基类都会有一个或多个虚函数。

bit (位) 计算机中基本信息单元。一个位的值可以是 0 或 1。

bug (程序漏洞) 程序中的错误。

byte (字节) 大多数计算机中的基本寻址单元。一个字节通常包含 8 位。

class (类) 用户自定义类型，可以包含数据成员、函数成员和成员类型。

code (代码) 程序或程序的一部分；既可表示源代码也可表示目标代码，从这个角度来说有二义性。

compiler (编译器) 将源代码转换为二进制代码的程序。

complexity (复杂性) 描述问题求解方案构造难度或方案本身难度的概念和衡量标准，很难准确定义。有时复杂性（简单）表示为执行算法所需的操作次数估计。

computation (计算) 执行某段代码，通常接受一些输入并生成一些输出。

concept (概念) (1) 一种概念（notion）、一种思想；(2) 一组要求，通常用于模板实参。

concrete class (具体类) 可创建对象的类。

constant (常量) (在给定作用域中) 不能改变的值；不可变。

constructor (构造函数) 初始化（“构造”）对象的操作。通常一个构造函数会建立一个不变

式，并且通常会获取对象要使用的资源（通常由析构函数释放这些资源）。

container(容器) 保存元素（其他对象）的对象。

copy (拷贝) 令两个对象具有相同值的操作。参见 **move (移动)**。

correctness (正确性) 若一个程序或一段程序满足其说明，则它是正确的。不幸的是，说明可能不完整或不一致，又或是无法满足用户的合理期望。因此，为了生成可接受的代码，我们有时不得不比形式化说明做得更多。

cost (代价) 生成一个程序或执行它的花费（如编程时间、运行时间或空间）。理想情况下，代价应该是复杂性的一个函数。

data (数据) 计算中用到的值。

debugging (调试) 在程序中查找并消除错误的工作；通常远没有测试那么系统化。

declaration (声明) 在程序中关于一个名字具有某个类型的说明。

definition (定义) 一个实体的声明，提供了所有必要信息，令程序可使用此实体。一种更简单的说法：分配了内存的声明。

derived class (派生类) 派生自一个或多个基类的类。

design (设计) 关于一个软件应该如何操作以满足其说明的总体描述。

destructor (析构函数) 当对象被销毁时（例如在作用域尾）被隐式调用的操作。它通常释放资源。

encapsulation (封装) 保护想要为私有性质的东西（如实现细节）不被未授权用户访问。

error (错误) 对程序行为的合理期望（通常表达为要求或用户指南）与程序实际表现不吻合。

executable (可执行程序) 已准备好运行于计算机上的程序。

feature creep (特性膨胀) 向程序添加过多功能只是“以备万一”的倾向。

file (文件) 计算机中永久保存的信息的容器。

floating-point number (浮点数) 计算机对实数的一种近似，例如 7.93 和 10.73e-3。

function (函数) 一个命名的代码单元，可以从程序中其他部分调用它；计算的逻辑单元。

generic programming (泛型编程) 一种程序设

计风格，关注算法的设计和高效实现。一个泛型算法能正确用于所有满足其要求的实参类型，在 C++ 中，泛型编程通常使用模板。

handle (句柄) 一个类，允许用户通过其成员指针或引用访问另一个类。参见 **copy, move, resource**。

header (头文件) 一个包含声明的文件，其中的声明用于程序中不同部分共享接口。

hiding (隐藏) 防止一部分信息被直接看到或直接访问的动作。例如，嵌套（内层）作用域中的名字可以防止来自外层（包围）作用域中的相同名字被直接使用。

ideal (理想) 我们所追求的完美版本。通常我们不得不做出折中、寻求近似版本。

implementation (实现) (1) 编写、测试代码的工作；(2) 实现程序的代码。

infinite loop (无限循环) 终止条件永不为真的循环。参见 **iteration**。

infinite recursion (无限递归) 直到用于保存调用数据的机器内存都耗光也未结束的递归。这种递归实际上不会无限执行下去，而是会由于某种硬件错误而终止。

information hiding (信息隐藏) 分离接口和实现的活动，从而隐藏了不希望吸引用户注意的实现细节并提供了抽象。

initialize (初始化) 为对象赋予最初的（初始）值。

input (输入) 计算用到的值（如函数实参和从键盘敲入的字符）。

integer (整数) 一个整数，如 42 和 -99。

interface (接口) 一个声明或一组声明，指明了一段代码（如一个函数或一个类）如何被调用。

invariant (不变式) 在程序给定位置（可能有很多位置）必须始终为真的东西；通常用于描述一个对象的状态（一组值）或进入重复语句之前的循环的状态。

iteration (迭代) 重复执行一段代码的动作；参见 **recursion**。

iterator (迭代器) 标识序列中元素的对象。

library (库) 一组类型、函数、类等等，实现了 一组特性（抽象），这些特性会被很多程序用作其一部分。

lifetime (生命期) 从对象的初始化时间一直到其不可用的时间（离开作用域、被释放或程序终止）。

linker (连接器) 将目标代码文件和库组合为一个可执行文件的程序。

literal (字面值常量) 直接指出值的语法表示方式，例如 12 指明整型值“十二”。

loop (循环) 反复执行的一段代码；在 C++ 中通常是 for 语句或 while 语句。

move (移动) 将一个值从一个对象转移到另一个对象的操作，原对象将获得表示“空”的值。参见 copy。

mutable (可变的) 可改变的；与不可变、常量相对，变量。

object (对象) (1) 已知类型、已初始化的一块内存，保存该类型的一个值；(2) 一块内存。

object code (目标代码) 编译器的输出，连接器的输入（连接器用它生成可执行代码）。

object file (目标文件) 包含目标代码的文件。

object-oriented programming (面向对象编程) 一种程序设计风格，关注类和类层次的设计和使用。

operation (操作) 用于执行某个动作，例如函数和运算符。

output (输出) 计算生成的值（如函数返回结果或写到显示屏的一行行字符）。

overflow (溢出) 生成的值不能保存到目标中。

overload (重载) 定义两个同名的函数或操作，但具有不同的参数（运算对象）类型。

override (覆盖) 在派生类中定义一个函数，与基类中的虚函数具有相同的名字和参数类型，因此可透过基类定义的接口被调用。

owner (所有者) 负责释放资源的对象。

paradigm (范型) 多少有些自命不凡的设计或编程风格术语；常常与（错误的）暗示一起使用——存在一种优于所有其他风格的范型。

parameter (参数) 函数或模板显式输入的声明。当被调用时，函数可以通过参数的名字访问传递来的实参。

pointer (指针) (1) 用于标识内存中带类型对象的值；(2) 保存这种值的变量。

post-condition (后置条件) 当退出一段代码（如

一个函数或一个循环）时必须满足的条件。

pre-condition (前置条件) 当进入一段代码（如一个函数或一个循环）时必须满足的条件。

program (程序) 足够完整可被计算机执行的代码（可能还有关联的数据）。

programming (程序设计或编程) 将问题求解方案表达为代码的艺术。

programming language (程序设计语言) 用于表达程序的语言。

pseudo code (伪代码) 计算的描述，用非正式表示方式而非程序设计语言书写。

pure virtual function (纯虚函数) 必须在派生类中覆盖的虚函数。

RAII (Resource Acquisition Is Initialization, 资源获取即初始化) 一种基于作用域的资源管理基本技术。

range (范围) 可以用一个起始点和一个结束点描述的值的序列。例如，[0:5) 表示值 0、1、2、3 和 4。

recursion (递归) 一个函数调用自身的动作；参见 iteration。

reference (引用) (1) 描述内存中一个带类型值的位置的值；(2) 保存这种值的变量。

regular expression (正则表达式) 字符串中模式的表示方式。

requirement (要求) (1) 程序或程序的一部分应具有行为的描述；(2) 函数或模板对其实参的假设的描述。

resource (资源) 需要获取并在随后释放的东西，例如文件句柄、锁或者内存。参见 handle、owner。

rounding (舍入) 一个值转换为数学上最接近的一个值，目标值的类型精确度更低。

scope (作用域) 程序文本（代码）的范围，给定名字在其中可被引用。

sequence (序列) 可线性顺序访问的一些元素。

software (软件) 一组代码段及相关数据；常常与 program 互换使用。

source code (源代码) 程序员生成的代码，（原则上）对其他程序员是可读的。

specification (说明) 一段代码应该做什么的说明。

standard (标准) 官方认可的某种东西的定义，例如一种程序设计语言。

state (状态) 一组值。

string (字符串) 字符序列。

style (风格) 一组程序设计技术，对语言特性的使用是一致的；有时会用于非常局限的意义——仅仅指代一些代码命名和外观的底层规则。

subtype (子类型) 派生类型；一个类型，具有另一个类型的所有属性，可能还更多。

supertype (超类型) 基类型；一个类型，具有另一个类型的属性的子集。

system (系统) (1) 一个程序或一组程序，在计算机上执行某个任务；(2) “操作系统”的简称，即计算机的基本执行环境和工具。

template (模板) 一个类或一个函数，用一个或多个类型或（编译时）值进行参数化；支持泛型编程的基本 C++ 语言结构。

testing (测试) 对程序错误的系统化查找。

trade-off (折中) 平衡多个设计和实现标准的结果。

truncation (截断) 从一个类型转换到另一个类型所产生的信息损失，原因是目标类型不能准确表达要转换的值。

type (类型) 定义了对象的一组可能取值及其一组操作的东西。

uninitialized (未初始化) 对象在初始化之前的（未定义的）状态。

unit (1) (单位) 标准度量，给值以含义（如表示距离的千米）；(2) (单元) 一个完整东西的与众不同的（如命名的）部分。

use case (用例) 程序的特殊的（通常是简单的）使用，用于测试其功能、展示其目的。

value (值) 内存中的一组位，根据类型解释其含义。

variable (变量) 给定类型的命名对象；除非未初始化，否则包含一个值。

virtual function (虚函数) 可在派生类中覆盖的成员函数。

word (字) 计算机内存的基本单元，通常用于保存一个整数的单元。

参 考 文 献

- Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (usually called “The Dragon Book”). Addison-Wesley, 2006. ISBN 0321486811.
- Andrews, Mike, and James A. Whittaker. *How to Break Software: Functional and Security Testing of Web Applications and Web Services*. Addison-Wesley, 2006. ISBN 0321369440.
- Bergin, Thomas J., and Richard G. Gibson, eds. *History of Programming Languages, Volume 2*. Addison-Wesley, 1996. ISBN 0201895021.
- Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. ISBN 0131872494.
- Boost.org. “A Repository for Libraries Meant to Work Well with the C++ Standard Library.” www.boost.org.
- Cox, Russ. “Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, . . .).” <http://swtch.com/~rsc/regexp/regexp1.html>.
- dmoz.org. <http://dmoz.org/Computers/Programming/Languages>.
- Freeman, T. L., and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992. ISBN 0136515975.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.
- Goldthwaite, Lois, ed. *Technical Report on C++ Performance*. ISO/IEC PDTR 18015. www.stroustrup.com/performanceTR.pdf.
- Gullberg, Jan. *Mathematics – From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X.
- Hailpern, Brent, and Barbara G. Ryder, eds. *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.
- ISO/IEC 9899:2011. *Programming Languages – C*. The C standard.
- ISO/IEC 14882:2011. *Programming Languages – C++*. The C++ standard.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988. ISBN 0131103628.
- Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1997. ISBN 0201896842.
- Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Langer, Angelika, and Klaus Kreft. *Standard C++ IOSTreams and Locales: Advanced Programmer’s Guide and Reference*. Addison-Wesley, 2000. ISBN 0321585585.
- Lippman, Stanley B., José Lajoie, and Barbara E. Moo. *The C++ Primer, Fifth Edition*. Addison-Wesley, 2005. ISBN 0321714113. (Use only the 5th edition.)
- Lockheed Martin Corporation. “Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program.” Document Number 2RDU00001 Rev C. December 2005. Colloquially known as “JSF++.” www.stroustrup.com/JSF-AV-rules.pdf.
- Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts – The Programmers Who Created the Software Revolution*. Basic Books, 2002. ISBN 978-0465042265.
- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001. ISBN 0201749629.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.
- Programming Research. *High-Integrity C++ Coding Standard Manual Version 2.4*. www.

- programmingresearch.com.
- Richards, Martin. *BCPL – The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 0521219655.
- Ritchie, Dennis. “The Development of the C Programming Language.” *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. ACM SIGPLAN Notices, Vol. 28 No. 3, 1993.
- Salus, Peter H. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.
- Sammet, Jean E. *Programming Languages: History and Fundamentals*. Prentice Hall, 1969. ISBN 0137299885.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2002. ISBN 0201604647.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003. ISBN 0201795256.
- Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition*. O'Reilly, 2005. ISBN 0596101058.
- Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.
- Sebesta, Robert W. *Concepts of Programming Languages, Sixth Edition*. Addison-Wesley, 2003. ISBN 0321193628.
- Shepherd, Simon. “The Tiny Encryption Algorithm (TEA).” www.tayloredge.com/reference/Mathematics/TEA-XTEA.pdf and <http://143.53.36.235:8080/tea.htm>.
- Stepanov, Alexander. www.stepanovpapers.com.
- Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.
- Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.
- Stroustrup, Bjarne. “A History of C++: 1979–1991.” *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. ACM SIGPLAN Notices, Vol. 28 No. 3, 1993.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. “Learning Standard C++ as a New Language.” *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. “C and C++: Siblings”; “C and C++: A Case for Compatibility”; and “C and C++: Case Studies in Compatibility.” *The C/C++ Users Journal*, July, Aug., and Sept. 2002.
- Stroustrup, Bjarne. “Evolving a Language in and for the Real World: C++ 1991–2006.” *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.
- Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.
- Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 978-0321958310.
- Stroustrup, Bjarne. Author’s home page, www.stroustrup.com.
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622.
- Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586.
- University of St. Andrews. The MacTutor History of Mathematics archive. <http://www-gap.dcs.st-and.ac.uk/~history>.
- Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.
- Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2002. ISBN 0201796198.
- Wood, Alastair. *Introduction to Numerical Analysis*. Addison-Wesley, 2000. ISBN 020134291X.

C++程序设计 原理与实践（基础篇）原书第2版

Programming Principles and Practice Using C++ Second Edition

《C++程序设计：原理与实践（原书第2版）》将经典程序设计思想与C++开发实践完美结合，全面地介绍了程序设计基本原理，包括基本概念、设计和编程技术、语言特性以及标准库等，教你学会如何编写具有输入、输出、计算以及简单图形显示等功能的程序。此外，本书通过对C++思想和历史的讨论、对经典实例（如矩阵运算、文本处理、测试以及嵌入式系统程序设计）的展示，以及对C语言的简单描述，为你呈现了一幅程序设计的全景图。

为方便读者循序渐进地学习，加上篇幅所限，《C++程序设计：原理与实践（原书第2版）》分为基础篇和进阶篇两册出版，基础篇包括第1~11章、第17~19章和附录A、C，进阶篇包括第12~16章、第20~27章和附录B、D、E。本书是基础篇。

本书特点

- **C++初学者的权威指南。**无论你是从事软件开发还是其他领域的工作，本书都将为你打开C++程序设计之门。
- **中高级程序员的必备参考。**通过观察程序设计大师如何处理编程中的各种问题，你将获得新的领悟和指引。
- **全面阐释C++基本概念和技术。**与传统的C++教材相比，本书对基本概念和技术的介绍更为深入，为你编写实用、正确、易维护和有效的代码打下坚实的基础。
- **强调现代C++（C++11和C++14）编程风格。**本书从开篇就介绍现代C++程序设计技术，并揭示了大量关于如何使用C++标准库以及C++11和C++14的特性来简化程序设计的原理，使你快速掌握实用编程技巧。
- **配套教辅资源丰富。**本书网站（www.stroustrup.com/Programming）提供了丰富的辅助资料，包括实例源码、PPT、勘误等。

作者简介

本贾尼·斯特劳斯特鲁普（Bjarne Stroustrup）英国剑桥大学计算机科学博士，C++的设计者和最初的实现者。他现在是德州农工大学计算机科学首席教授。1993年，由于在C++领域的重大贡献，他获得了ACM的Grace Murray Hopper大奖并成为ACM院士。在进入学术界之前，他在AT&T贝尔实验室工作，是ISO C++标准委员会的创始人之一。



Pearson

www.pearson.com

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259



华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn

上架指导：计算机\程序设计

ISBN 978-7-111-56225-2



9 787111 562252

定价：99.00元

[General Information]

书名=C++程序设计原理与实践(基础篇)(原书第2版)

作者=(美)本贾尼·斯特劳斯·鲁普著

页数=403

SS号=14211815

DX号=

出版日期=2017.04

出版社=机械工业出版社